Numerical Recipes
in Fortran 77

The Art of Scientific Computing

Second Edition

Volume 1 of
Fortran Numerical Recipes

William H. Press

Harvard-Smithsonian Center for Astrophysics

Saul A. Teukolsky

Department of Physics, Cornell University

William T. Vetterling

Polaroid Corporation

Brian P. Flannery

EXXON Research and Engineering Company

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

Copyright © Cambridge University Press 1986, 1992
except for §13.10, which is placed into the public domain,
and except for all other computer programs and procedures, which are
Copyright © Numerical Recipes Software 1986, 1992, 1997
All Rights Reserved.

Some sections of this book were originally published, in different form, in *Computers in Physics* magazine, Copyright © American Institute of Physics, 1988–1992.

First Edition originally published 1986; Second Edition originally published 1992 as *Numerical Recipes in FORTRAN: The Art of Scientific Computing*
Reprinted with corrections, 1993, 1994, 1995.
Reprinted with corrections, 1996, 1997, as *Numerical Recipes in Fortran 77: The Art of Scientific Computing* (Vol. 1 of Fortran Numerical Recipes)

This reprinting is corrected to software version 2.08

Printed in the United States of America
Typeset in TEX

Without an additional license to use the contained software, this book is intended as a text and reference book, for reading purposes only. A free license for limited use of the software by the individual owner of a copy of this book who personally types one or more routines into a single computer is granted under terms described on p. xxi. See the section "License Information" (pp. xx–xxiii) for information on obtaining more general licenses at low cost.

Machine-readable media containing the software in this book, with included licenses for use on a single screen, are available from Cambridge University Press. See the order form at the back of the book, email to "orders@cup.org" (North America) or "trade@cup.cam.ac.uk" (rest of world), or write to Cambridge University Press, 110 Midland Avenue, Port Chester, NY 10573 (USA), for further information.

The software may also be downloaded, with immediate purchase of a license also possible, from the Numerical Recipes Software Web Site (`http://www.nr.com`). Unlicensed transfer of Numerical Recipes programs to any other format, or to any computer except one that is specifically licensed, is strictly prohibited. Technical questions, corrections, and requests for information should be addressed to Numerical Recipes Software, P.O. Box 243, Cambridge, MA 02238 (USA), email "info@nr.com", or fax 781 863-1739.

Library of Congress Cataloging in Publication Data

Numerical recipes in Fortran 77 : the art of scientific computing / William H. Press ... [et al.]. – 2nd ed.

Includes bibliographical references (p.) and index.

ISBN 0-521-43064-X

1. Numerical analysis–Computer programs. 2. Science–Mathematics–Computer programs. 3. FORTRAN (Computer program language) I. Press, William H.
QA297.N866 1992
519.4′0285′53–dc20 92-8876

A catalog record for this book is available from the British Library.

ISBN 0 521 43064 X Volume 1 (this book)
ISBN 0 521 57439 0 Volume 2
ISBN 0 521 43721 0 Example book in FORTRAN
ISBN 0 521 57440 4 FORTRAN diskette (IBM 3.5″)
ISBN 0 521 57608 3 CDROM (IBM PC/Macintosh)
ISBN 0 521 57607 5 CDROM (UNIX)

Contents

Plan of the Two-Volume Edition ***xiii***

Preface to the Second Edition ***xv***

Preface to the First Edition ***xviii***

License Information ***xx***

Computer Programs by Chapter and Section ***xxiv***

1 Preliminaries ***1***

1.0 Introduction 1
1.1 Program Organization and Control Structures 5
1.2 Error, Accuracy, and Stability 18

2 Solution of Linear Algebraic Equations ***22***

2.0 Introduction 22
2.1 Gauss-Jordan Elimination 27
2.2 Gaussian Elimination with Backsubstitution 33
2.3 LU Decomposition and Its Applications 34
2.4 Tridiagonal and Band Diagonal Systems of Equations 42
2.5 Iterative Improvement of a Solution to Linear Equations 47
2.6 Singular Value Decomposition 51
2.7 Sparse Linear Systems 63
2.8 Vandermonde Matrices and Toeplitz Matrices 82
2.9 Cholesky Decomposition 89
2.10 QR Decomposition 91
2.11 Is Matrix Inversion an N^3 Process? 95

3 Interpolation and Extrapolation ***99***

3.0 Introduction 99
3.1 Polynomial Interpolation and Extrapolation 102
3.2 Rational Function Interpolation and Extrapolation 104
3.3 Cubic Spline Interpolation 107
3.4 How to Search an Ordered Table 110
3.5 Coefficients of the Interpolating Polynomial 113
3.6 Interpolation in Two or More Dimensions 116

4 Integration of Functions 123

4.0 Introduction 123
4.1 Classical Formulas for Equally Spaced Abscissas 124
4.2 Elementary Algorithms 130
4.3 Romberg Integration 134
4.4 Improper Integrals 135
4.5 Gaussian Quadratures and Orthogonal Polynomials 140
4.6 Multidimensional Integrals 155

5 Evaluation of Functions 159

5.0 Introduction 159
5.1 Series and Their Convergence 159
5.2 Evaluation of Continued Fractions 163
5.3 Polynomials and Rational Functions 167
5.4 Complex Arithmetic 171
5.5 Recurrence Relations and Clenshaw's Recurrence Formula 172
5.6 Quadratic and Cubic Equations 178
5.7 Numerical Derivatives 180
5.8 Chebyshev Approximation 184
5.9 Derivatives or Integrals of a Chebyshev-approximated Function 189
5.10 Polynomial Approximation from Chebyshev Coefficients 191
5.11 Economization of Power Series 192
5.12 Padé Approximants 194
5.13 Rational Chebyshev Approximation 197
5.14 Evaluation of Functions by Path Integration 201

6 Special Functions 205

6.0 Introduction 205
6.1 Gamma Function, Beta Function, Factorials, Binomial Coefficients 206
6.2 Incomplete Gamma Function, Error Function, Chi-Square Probability Function, Cumulative Poisson Function 209
6.3 Exponential Integrals 215
6.4 Incomplete Beta Function, Student's Distribution, F-Distribution, Cumulative Binomial Distribution 219
6.5 Bessel Functions of Integer Order 223
6.6 Modified Bessel Functions of Integer Order 229
6.7 Bessel Functions of Fractional Order, Airy Functions, Spherical Bessel Functions 234
6.8 Spherical Harmonics 246
6.9 Fresnel Integrals, Cosine and Sine Integrals 248
6.10 Dawson's Integral 252
6.11 Elliptic Integrals and Jacobian Elliptic Functions 254
6.12 Hypergeometric Functions 263

7 Random Numbers 266

7.0 Introduction 266
7.1 Uniform Deviates 267

7.2 Transformation Method: Exponential and Normal Deviates 277
7.3 Rejection Method: Gamma, Poisson, Binomial Deviates 281
7.4 Generation of Random Bits 287
7.5 Random Sequences Based on Data Encryption 290
7.6 Simple Monte Carlo Integration 295
7.7 Quasi- (that is, Sub-) Random Sequences 299
7.8 Adaptive and Recursive Monte Carlo Methods 306

8 Sorting 320

8.0 Introduction 320
8.1 Straight Insertion and Shell's Method 321
8.2 Quicksort 323
8.3 Heapsort 327
8.4 Indexing and Ranking 329
8.5 Selecting the Mth Largest 333
8.6 Determination of Equivalence Classes 337

9 Root Finding and Nonlinear Sets of Equations 340

9.0 Introduction 340
9.1 Bracketing and Bisection 343
9.2 Secant Method, False Position Method, and Ridders' Method 347
9.3 Van Wijngaarden–Dekker–Brent Method 352
9.4 Newton-Raphson Method Using Derivative 355
9.5 Roots of Polynomials 362
9.6 Newton-Raphson Method for Nonlinear Systems of Equations 372
9.7 Globally Convergent Methods for Nonlinear Systems of Equations 376

10 Minimization or Maximization of Functions 387

10.0 Introduction 387
10.1 Golden Section Search in One Dimension 390
10.2 Parabolic Interpolation and Brent's Method in One Dimension 395
10.3 One-Dimensional Search with First Derivatives 399
10.4 Downhill Simplex Method in Multidimensions 402
10.5 Direction Set (Powell's) Methods in Multidimensions 406
10.6 Conjugate Gradient Methods in Multidimensions 413
10.7 Variable Metric Methods in Multidimensions 418
10.8 Linear Programming and the Simplex Method 423
10.9 Simulated Annealing Methods 436

11 Eigensystems 449

11.0 Introduction 449
11.1 Jacobi Transformations of a Symmetric Matrix 456
11.2 Reduction of a Symmetric Matrix to Tridiagonal Form: Givens and Householder Reductions 462
11.3 Eigenvalues and Eigenvectors of a Tridiagonal Matrix 469
11.4 Hermitian Matrices 475
11.5 Reduction of a General Matrix to Hessenberg Form 476

11.6 The QR Algorithm for Real Hessenberg Matrices 480
11.7 Improving Eigenvalues and/or Finding Eigenvectors by Inverse Iteration 487

12 Fast Fourier Transform 490

12.0 Introduction 490
12.1 Fourier Transform of Discretely Sampled Data 494
12.2 Fast Fourier Transform (FFT) 498
12.3 FFT of Real Functions, Sine and Cosine Transforms 504
12.4 FFT in Two or More Dimensions 515
12.5 Fourier Transforms of Real Data in Two and Three Dimensions 519
12.6 External Storage or Memory-Local FFTs 525

13 Fourier and Spectral Applications 530

13.0 Introduction 530
13.1 Convolution and Deconvolution Using the FFT 531
13.2 Correlation and Autocorrelation Using the FFT 538
13.3 Optimal (Wiener) Filtering with the FFT 539
13.4 Power Spectrum Estimation Using the FFT 542
13.5 Digital Filtering in the Time Domain 551
13.6 Linear Prediction and Linear Predictive Coding 557
13.7 Power Spectrum Estimation by the Maximum Entropy (All Poles) Method 565
13.8 Spectral Analysis of Unevenly Sampled Data 569
13.9 Computing Fourier Integrals Using the FFT 577
13.10 Wavelet Transforms 584
13.11 Numerical Use of the Sampling Theorem 600

14 Statistical Description of Data 603

14.0 Introduction 603
14.1 Moments of a Distribution: Mean, Variance, Skewness, and So Forth 604
14.2 Do Two Distributions Have the Same Means or Variances? 609
14.3 Are Two Distributions Different? 614
14.4 Contingency Table Analysis of Two Distributions 622
14.5 Linear Correlation 630
14.6 Nonparametric or Rank Correlation 633
14.7 Do Two-Dimensional Distributions Differ? 640
14.8 Savitzky-Golay Smoothing Filters 644

15 Modeling of Data 650

15.0 Introduction 650
15.1 Least Squares as a Maximum Likelihood Estimator 651
15.2 Fitting Data to a Straight Line 655
15.3 Straight-Line Data with Errors in Both Coordinates 660
15.4 General Linear Least Squares 665
15.5 Nonlinear Models 675

15.6 Confidence Limits on Estimated Model Parameters 684
15.7 Robust Estimation 694

16 Integration of Ordinary Differential Equations 701

16.0 Introduction 701
16.1 Runge-Kutta Method 704
16.2 Adaptive Stepsize Control for Runge-Kutta 708
16.3 Modified Midpoint Method 716
16.4 Richardson Extrapolation and the Bulirsch-Stoer Method 718
16.5 Second-Order Conservative Equations 726
16.6 Stiff Sets of Equations 727
16.7 Multistep, Multivalue, and Predictor-Corrector Methods 740

17 Two Point Boundary Value Problems 745

17.0 Introduction 745
17.1 The Shooting Method 749
17.2 Shooting to a Fitting Point 751
17.3 Relaxation Methods 753
17.4 A Worked Example: Spheroidal Harmonics 764
17.5 Automated Allocation of Mesh Points 774
17.6 Handling Internal Boundary Conditions or Singular Points 775

18 Integral Equations and Inverse Theory 779

18.0 Introduction 779
18.1 Fredholm Equations of the Second Kind 782
18.2 Volterra Equations 786
18.3 Integral Equations with Singular Kernels 788
18.4 Inverse Problems and the Use of A Priori Information 795
18.5 Linear Regularization Methods 799
18.6 Backus-Gilbert Method 806
18.7 Maximum Entropy Image Restoration 809

19 Partial Differential Equations 818

19.0 Introduction 818
19.1 Flux-Conservative Initial Value Problems 825
19.2 Diffusive Initial Value Problems 838
19.3 Initial Value Problems in Multidimensions 844
19.4 Fourier and Cyclic Reduction Methods for Boundary Value Problems 848
19.5 Relaxation Methods for Boundary Value Problems 854
19.6 Multigrid Methods for Boundary Value Problems 862

20 Less-Numerical Algorithms 881

20.0 Introduction 881
20.1 Diagnosing Machine Parameters 881
20.2 Gray Codes 886

20.3 Cyclic Redundancy and Other Checksums 888
20.4 Huffman Coding and Compression of Data 896
20.5 Arithmetic Coding 902
20.6 Arithmetic at Arbitrary Precision 906

References for Volume 1 ***916***

Index of Programs and Dependencies (Vol. 1) ***921***

General Index to Volumes 1 and 2

Contents of Volume 2: Numerical Recipes in Fortran 90

Preface to Volume 2 ***viii***
Foreword by Michael Metcalf ***x***
License Information ***xvii***

21 Introduction to Fortran 90 Language Features ***935***
22 Introduction to Parallel Programming ***962***
23 Numerical Recipes Utilities for Fortran 90 ***987***

Fortran 90 Code Chapters ***1009***

B1 Preliminaries ***1010***
B2 Solution of Linear Algebraic Equations ***1014***
B3 Interpolation and Extrapolation ***1043***
B4 Integration of Functions ***1052***
B5 Evaluation of Functions ***1070***
B6 Special Functions ***1083***
B7 Random Numbers ***1141***
B8 Sorting ***1167***
B9 Root Finding and Nonlinear Sets of Equations ***1182***
B10 Minimization or Maximization of Functions ***1201***
B11 Eigensystems ***1225***
B12 Fast Fourier Transform ***1235***

B13 Fourier and Spectral Applications 1253
B14 Statistical Description of Data 1269
B15 Modeling of Data 1285
B16 Integration of Ordinary Differential Equations 1297
B17 Two Point Boundary Value Problems 1314
B18 Integral Equations and Inverse Theory 1325
B19 Partial Differential Equations 1332
B20 Less-Numerical Algorithms 1343

References for Volume 2 1359

Appendices

C1 Listing of Utility Modules (nrtype and nrutil) 1361
C2 Listing of Explicit Interfaces 1384
C3 Index of Programs and Dependencies (Vol. 2) 1434

General Index to Volumes 1 and 2 1447

Plan of the Two-Volume Edition

Fortran, long the epitome of stability, is once again a language in flux. Fortran 90 is not just the long-awaited updating of traditional Fortran 77 to modern computing practices, but also demonstrates Fortran's decisive bid to be the language of choice for parallel programming on multiprocessor computers.

At the same time, Fortran 90 is completely backwards-compatible with all Fortran 77 code. So, users with legacy code, or who choose to use only older language constructs, will still get the benefit of updated and actively maintained compilers.

As we, the authors of *Numerical Recipes*, watched the gestation and birth of Fortran 90 by its governing standards committee (an interesting process described by a leading Committee member, Michael Metcalf, in the Foreword to our Volume 2), it became clear to us that the right moment for moving Numerical Recipes from Fortran 77 to Fortran 90 was sooner, rather than later.

On the other hand, it was equally clear that Fortran-77-style programming — no matter whether with Fortran 77 or Fortran 90 compilers — is, and will continue for a long time to be, the "mother tongue" of a large population of active scientists, engineers, and other users of numerical computation. This is not a user base that we would willingly or knowingly abandon.

The solution was immediately clear: a two-volume edition of the Fortran *Numerical Recipes* consisting of Volume 1 (this one, a corrected reprinting of the previous one-volume edition), now retitled *Numerical Recipes in Fortran 77*, and a completely new Volume 2, titled *Numerical Recipes in Fortran 90: The Art of* Parallel *Scientific Computing*. Volume 2 begins with three chapters (21, 22, and 23) that extend the narrative of the first volume to the new subjects of Fortran 90 language features, parallel programming methodology, and the implementation of certain useful utility functions in Fortran 90. Then, in exact correspondence with Volume 1's Chapters 1–20, are new chapters B1–B20, devoted principally to the listing and explanation of new Fortran 90 routines. With a few exceptions, each Fortran 77 routine in Volume 1 has a corresponding new Fortran 90 version in Volume 2. (The exceptions are a few new capabilities, notably in random number generation and in multigrid PDE solvers, that are unique to Volume 2's Fortran 90.) Otherwise, there is no duplication between the volumes. The detailed explanation of the algorithms in this Volume 1 is intended to apply to, and be essential for, both volumes.

In other words: **You can use this Volume 1 without having Volume 2, but you can't use Volume 2 without Volume 1.** We think that there is much to be gained by having and using *both* volumes: Fortran 90's parallel language constructions are not only useful for present and future multiprocessor machines; they also allow for the elegant and concise formulation of many algorithms on ordinary single-processor computers. We think that essentially *all* Fortran programmers will want gradually to migrate into Fortran 90 and into a mode of "thinking parallel." We have written Volume 2 specifically to help with this important transition.

Volume 2's discussion of parallel programming is focused on those issues of direct relevance to the Fortran 90 programmer. Some more general aspects of parallel programming, such as communication costs, synchronization of multiple processers,

etc., are touched on only briefly. We provide references to the extensive literature on these more specialized topics.

A special note to C programmers: Right now, there is no effort at producing a parallel version of C that is comparable to Fortran 90 in maturity, acceptance, and stability. We think, therefore, that C programmers will be well served by using Volume 2, either in conjuction with this Volume 1 or else in conjunction with the sister volume *Numerical Recipes in C: The Art of Scientific Computing*, for an educational excursion into Fortran 90, its parallel programming constructions, and the numerical algorithms that capitalize on them. C and C++ programming have not been far from our minds as we have written this two-volume version. We think you will find that time spent in absorbing the principal lessons of Volume 2's Chapters 21–23 will be amply repaid in the future, as C and C++ eventually develop standard parallel extensions.

Preface to the Second Edition

Our aim in writing the original edition of *Numerical Recipes* was to provide a book that combined general discussion, analytical mathematics, algorithmics, and actual working programs. The success of the first edition puts us now in a difficult, though hardly unenviable, position. We wanted, then and now, to write a book that is informal, fearlessly editorial, unesoteric, and above all useful. There is a danger that, if we are not careful, we might produce a second edition that is weighty, balanced, scholarly, and boring.

It is a mixed blessing that we know more now than we did six years ago. Then, we were making educated guesses, based on existing literature and our own research, about which numerical techniques were the most important and robust. Now, we have the benefit of direct feedback from a large reader community. Letters to our alter-ego enterprise, Numerical Recipes Software, are in the thousands per year. (Please, *don't telephone* us.) Our post office box has become a magnet for letters pointing out that we have omitted some particular technique, well known to be important in a particular field of science or engineering. We value such letters, and digest them carefully, especially when they point us to specific references in the literature.

The inevitable result of this input is that this Second Edition of *Numerical Recipes* is substantially larger than its predecessor, in fact about 50% larger both in words and number of included programs (the latter now numbering well over 300). "Don't let the book grow in size," is the advice that we received from several wise colleagues. We have tried to follow the intended spirit of that advice, even as we violate the letter of it. We have not lengthened, or increased in difficulty, the book's principal discussions of mainstream topics. Many new topics are presented at this same accessible level. Some topics, both from the earlier edition and new to this one, are now set in smaller type that labels them as being "advanced." The reader who ignores such advanced sections completely will not, we think, find any lack of continuity in the shorter volume that results.

Here are some highlights of the new material in this Second Edition:

- a new chapter on integral equations and inverse methods
- a detailed treatment of multigrid methods for solving elliptic partial differential equations
- routines for band diagonal linear systems
- improved routines for linear algebra on sparse matrices
- Cholesky and QR decomposition
- orthogonal polynomials and Gaussian quadratures for arbitrary weight functions
- methods for calculating numerical derivatives
- Padé approximants, and rational Chebyshev approximation
- Bessel functions, and modified Bessel functions, of fractional order; and several other new special functions
- improved random number routines
- quasi-random sequences
- routines for adaptive and recursive Monte Carlo integration in high-dimensional spaces
- globally convergent methods for sets of nonlinear equations

- simulated annealing minimization for continuous control spaces
- fast Fourier transform (FFT) for real data in two and three dimensions
- fast Fourier transform (FFT) using external storage
- improved fast cosine transform routines
- wavelet transforms
- Fourier integrals with upper and lower limits
- spectral analysis on unevenly sampled data
- Savitzky-Golay smoothing filters
- fitting straight line data with errors in both coordinates
- a two-dimensional Kolmogorov-Smirnoff test
- the statistical bootstrap method
- embedded Runge-Kutta-Fehlberg methods for differential equations
- high-order methods for stiff differential equations
- a new chapter on "less-numerical" algorithms, including Huffman and arithmetic coding, arbitrary precision arithmetic, and several other topics.

Consult the Preface to the First Edition, following, or the Table of Contents, for a list of the more "basic" subjects treated.

Acknowledgments

It is not possible for us to list by name here all the readers who have made useful suggestions; we are grateful for these. In the text, we attempt to give specific attribution for ideas that appear to be original, and not known in the literature. We apologize in advance for any omissions.

Some readers and colleagues have been particularly generous in providing us with ideas, comments, suggestions, and programs for this Second Edition. We especially want to thank George Rybicki, Philip Pinto, Peter Lepage, Robert Lupton, Douglas Eardley, Ramesh Narayan, David Spergel, Alan Oppenheim, Sallie Baliunas, Scott Tremaine, Glennys Farrar, Steven Block, John Peacock, Thomas Loredo, Matthew Choptuik, Gregory Cook, L. Samuel Finn, P. Deuflhard, Harold Lewis, Peter Weinberger, David Syer, Richard Ferch, Steven Ebstein, and William Gould. We have been helped by Nancy Lee Snyder's mastery of a complicated TEX manuscript. We express appreciation to our editors Lauren Cowles and Alan Harvey at Cambridge University Press, and to our production editor Russell Hahn. We remain, of course, grateful to the individuals acknowledged in the Preface to the First Edition.

Special acknowledgment is due to programming consultant Seth Finkelstein, who influenced many of the routines in this book, and wrote or rewrote many more routines in its C-language twin and the companion Example books. Our project has benefited enormously from Seth's talent for detecting, and following the trail of, even very slight anomalies (often compiler bugs, but occasionally our errors), and from his good programming sense.

We prepared this book for publication on DEC and Sun workstations running the UNIX operating system, and on a 486/33 PC compatible running MS-DOS 5.0/Windows 3.0. (See §1.0 for a list of additional computers used in program tests.) We enthusiastically recommend the principal software used: GNU Emacs, TEX, Perl, Adobe Illustrator, and PostScript. Also used were a variety of FORTRAN compilers — too numerous (and sometimes too buggy) for individual

acknowledgment. It is a sobering fact that our standard test suite (exercising all the routines in this book) has uncovered compiler bugs in a large majority of the compilers tried. When possible, we work with developers to see that such bugs get fixed; we encourage interested compiler developers to contact us about such arrangements.

WHP and SAT acknowledge the continued support of the U.S. National Science Foundation for their research on computational methods. D.A.R.P.A. support is acknowledged for §13.10 on wavelets.

June, 1992

William H. Press
Saul A. Teukolsky
William T. Vetterling
Brian P. Flannery

Preface to the First Edition

We call this book *Numerical Recipes* for several reasons. In one sense, this book is indeed a "cookbook" on numerical computation. However there is an important distinction between a cookbook and a restaurant menu. The latter presents choices among complete dishes in each of which the individual flavors are blended and disguised. The former — and this book — reveals the individual ingredients and explains how they are prepared and combined.

Another purpose of the title is to connote an eclectic mixture of presentational techniques. This book is unique, we think, in offering, for each topic considered, a certain amount of general discussion, a certain amount of analytical mathematics, a certain amount of discussion of algorithmics, and (most important) actual implementations of these ideas in the form of working computer routines. Our task has been to find the right balance among these ingredients for each topic. You will find that for some topics we have tilted quite far to the analytic side; this where we have felt there to be gaps in the "standard" mathematical training. For other topics, where the mathematical prerequisites are universally held, we have tilted towards more in-depth discussion of the nature of the computational algorithms, or towards practical questions of implementation.

We admit, therefore, to some unevenness in the "level" of this book. About half of it is suitable for an advanced undergraduate course on numerical computation for science or engineering majors. The other half ranges from the level of a graduate course to that of a professional reference. Most cookbooks have, after all, recipes at varying levels of complexity. An attractive feature of this approach, we think, is that the reader can use the book at increasing levels of sophistication as his/her experience grows. Even inexperienced readers should be able to use our most advanced routines as black boxes. Having done so, we hope that these readers will subsequently go back and learn what secrets are inside.

If there is a single dominant theme in this book, it is that practical methods of numerical computation can be simultaneously efficient, clever, and — important — clear. The alternative viewpoint, that efficient computational methods must necessarily be so arcane and complex as to be useful only in "black box" form, we firmly reject.

Our purpose in this book is thus to open up a large number of computational black boxes to your scrutiny. We want to teach you to take apart these black boxes and to put them back together again, modifying them to suit your specific needs. We assume that you are mathematically literate, i.e., that you have the normal mathematical preparation associated with an undergraduate degree in a physical science, or engineering, or economics, or a quantitative social science. We assume that you know how to program a computer. We do not assume that you have any prior formal knowledge of numerical analysis or numerical methods.

The scope of *Numerical Recipes* is supposed to be "everything up to, but not including, partial differential equations." We honor this in the breach: First, we *do* have one introductory chapter on methods for partial differential equations (Chapter 19). Second, we obviously cannot include *everything* else. All the so-called "standard" topics of a numerical analysis course have been included in this book:

linear equations (Chapter 2), interpolation and extrapolation (Chaper 3), integration (Chaper 4), nonlinear root-finding (Chapter 9), eigensystems (Chapter 11), and ordinary differential equations (Chapter 16). Most of these topics have been taken beyond their standard treatments into some advanced material which we have felt to be particularly important or useful.

Some other subjects that we cover in detail are not usually found in the standard numerical analysis texts. These include the evaluation of functions and of particular special functions of higher mathematics (Chapters 5 and 6); random numbers and Monte Carlo methods (Chapter 7); sorting (Chapter 8); optimization, including multidimensional methods (Chapter 10); Fourier transform methods, including FFT methods and other spectral methods (Chapters 12 and 13); two chapters on the statistical description and modeling of data (Chapters 14 and 15); and two-point boundary value problems, both shooting and relaxation methods (Chapter 17).

The programs in this book are included in ANSI-standard `FORTRAN`-77. Versions of the book in `C`, `Pascal`, and `BASIC` are available separately. We have more to say about the `FORTRAN` language, and the computational environment assumed by our routines, in §1.1 (Introduction).

Acknowledgments

Many colleagues have been generous in giving us the benefit of their numerical and computational experience, in providing us with programs, in commenting on the manuscript, or in general encouragement. We particularly wish to thank George Rybicki, Douglas Eardley, Philip Marcus, Stuart Shapiro, Paul Horowitz, Bruce Musicus, Irwin Shapiro, Stephen Wolfram, Henry Abarbanel, Larry Smarr, Richard Muller, John Bahcall, and A.G.W. Cameron.

We also wish to acknowledge two individuals whom we have never met: Forman Acton, whose 1970 textbook *Numerical Methods that Work* (New York: Harper and Row) has surely left its stylistic mark on us; and Donald Knuth, both for his series of books on *The Art of Computer Programming* (Reading, MA: Addison-Wesley), and for TEX, the computer typesetting language which immensely aided production of this book.

Research by the authors on computational methods was supported in part by the U.S. National Science Foundation.

October, 1985

William H. Press
Brian P. Flannery
Saul A. Teukolsky
William T. Vetterling

License Information

Read this section if you want to use the programs in this book on a computer. You'll need to read the following Disclaimer of Warranty, get the programs onto your computer, and acquire a Numerical Recipes software license. (Without this license, which can be the free "immediate license" under terms described below, the book is intended as a text and reference book, for reading purposes only.)

Disclaimer of Warranty

We make no warranties, express or implied, that the programs contained in this volume are free of error, or are consistent with any particular standard of merchantability, or that they will meet your requirements for any particular application. They should not be relied on for solving a problem whose incorrect solution could result in injury to a person or loss of property. If you do use the programs in such a manner, it is at your own risk. The authors and publisher disclaim all liability for direct or consequential damages resulting from your use of the programs.

How to Get the Code onto Your Computer

Pick one of the following methods:

- You can type the programs from this book directly into your computer. In this case, the *only* kind of license available to you is the free "immediate license" (see below). You are not authorized to transfer or distribute a machine-readable copy to any other person, nor to have any other person type the programs into a computer on your behalf. We do not want to hear bug reports from you if you choose this option, because experience has shown that *virtually all* reported bugs in such cases are typing errors!

- You can download the Numerical Recipes programs electronically from the Numerical Recipes On-Line Software Store, located at our Web site (`http://www.nr.com`). They are packaged as a password-protected file, and you'll need to purchase a license to unpack them. You can get a single-screen license and password immediately, on-line, from the On-Line Store, with fees ranging from $50 (PC, Macintosh, educational institutions' UNIX) to $140 (general UNIX). Downloading the packaged software from the On-Line Store is also the way to start if you want to acquire a more general (multiscreen, site, or corporate) license.

- You can purchase media containing the programs from Cambridge University Press. Diskette versions are available in IBM-compatible format for machines running Windows 3.1, 95, or NT. CDROM versions in ISO-9660 format for PC, Macintosh, and UNIX systems are also available; these include both Fortran and C versions (as well as versions in Pascal and BASIC from the first edition) on a single CDROM. Diskettes purchased from Cambridge University Press include a single-screen license for PC or Macintosh only. The CDROM is available with a single-screen license for PC or Macintosh (order ISBN 0 521 576083), or (at a slightly higher price) with a single-screen license for UNIX workstations (order ISBN 0 521 576075). Orders for media from Cambridge University Press can be placed at 800 872-7423 (North America only) or by email to orders@cup.org (North America) or trade@cup.cam.ac.uk (rest of world). Or, visit the Web sites `http://www.cup.org` (North America) or `http://www.cup.cam.ac.uk` (rest of world).

Types of License Offered

Here are the types of licenses that we offer. Note that some types are automatically acquired with the purchase of media from Cambridge University Press, or of an unlocking password from the Numerical Recipes On-Line Software Store, while other types of licenses require that you communicate specifically with Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739). Our Web site `http://www.nr.com` has additional information.

- ["Immediate License"] If you are the individual owner of a copy of this book and you type one or more of its routines into your computer, we authorize you to use them on that computer for your own personal and noncommercial purposes. You are not authorized to transfer or distribute machine-readable copies to any other person, or to use the routines on more than one machine, or to distribute executable programs containing our routines. This is the only free license.
- ["Single-Screen License"] This is the most common type of low-cost license, with terms governed by our Single Screen (Shrinkwrap) License document (complete terms available through our Web site). Basically, this license lets you use Numerical Recipes routines on any one screen (PC, workstation, X-terminal, etc.). You may also, under this license, transfer pre-compiled, executable programs incorporating our routines to other, unlicensed, screens or computers, providing that (i) your application is noncommercial (i.e., does not involve the selling of your program for a fee), (ii) the programs were first developed, compiled, and successfully run on a licensed screen, and (iii) our routines are bound into the programs in such a manner that they cannot be accessed as individual routines and cannot practicably be unbound and used in other programs. That is, under this license, your program user must not be able to use our programs as part of a program library or "mix-and-match" workbench. Conditions for other types of commercial or noncommercial distribution may be found on our Web site (`http://www.nr.com`).

- ["Multi-Screen, Server, Site, and Corporate Licenses"] The terms of the Single Screen License can be extended to designated groups of machines, defined by number of screens, number of machines, locations, or ownership. Significant discounts from the corresponding single-screen prices are available when the estimated number of screens exceeds 40. Contact Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739) for details.
- ["Course Right-to-Copy License"] Instructors at accredited educational institutions who have adopted this book for a course, and who have already purchased a Single Screen License (either acquired with the purchase of media, or from the Numerical Recipes On-Line Software Store), may license the programs for use in that course as follows: Mail your name, title, and address; the course name, number, dates, and estimated enrollment; and advance payment of $5 per (estimated) student to Numerical Recipes Software, at this address: P.O. Box 243, Cambridge, MA 02238 (USA). You will receive by return mail a license authorizing you to make copies of the programs for use by your students, and/or to transfer the programs to a machine accessible to your students (but only for the duration of the course).

About Copyrights on Computer Programs

Like artistic or literary compositions, computer programs are protected by copyright. Generally it is an infringement for you to copy into your computer a program from a copyrighted source. (It is also not a friendly thing to do, since it deprives the program's author of compensation for his or her creative effort.) Under copyright law, all "derivative works" (modified versions, or translations into another computer language) also come under the same copyright as the original work.

Copyright does not protect ideas, but only the expression of those ideas in a particular form. In the case of a computer program, the ideas consist of the program's methodology and algorithm, including the necessary sequence of steps adopted by the programmer. The expression of those ideas is the program source code (particularly any arbitrary or stylistic choices embodied in it), its derived object code, and any other derivative works.

If you analyze the ideas contained in a program, and then express those ideas in your own completely different implementation, then that new program implementation belongs to you. That is what we have done for those programs in this book that are not entirely of our own devising. When programs in this book are said to be "based" on programs published in copyright sources, we mean that the ideas are the same. The expression of these ideas as source code is our own. We believe that no material in this book infringes on an existing copyright.

Trademarks

Several registered trademarks appear within the text of this book: Sun is a trademark of Sun Microsystems, Inc. SPARC and SPARCstation are trademarks of SPARC International, Inc. Microsoft, Windows 95, Windows NT, PowerStation, and MS are trademarks of Microsoft Corporation. DEC, VMS, Alpha AXP, and

ULTRIX are trademarks of Digital Equipment Corporation. IBM is a trademark of International Business Machines Corporation. Apple and Macintosh are trademarks of Apple Computer, Inc. UNIX is a trademark licensed exclusively through X/Open Co. Ltd. IMSL is a trademark of Visual Numerics, Inc. NAG refers to proprietary computer software of Numerical Algorithms Group (USA) Inc. PostScript and Adobe Illustrator are trademarks of Adobe Systems Incorporated. Last, and no doubt least, Numerical Recipes (when identifying products) is a trademark of Numerical Recipes Software.

Attributions

The fact that ideas are legally "free as air" in no way supersedes the ethical requirement that ideas be credited to their known originators. When programs in this book are based on known sources, whether copyrighted or in the public domain, published or "handed-down," we have attempted to give proper attribution. Unfortunately, the lineage of many programs in common circulation is often unclear. We would be grateful to readers for new or corrected information regarding attributions, which we will attempt to incorporate in subsequent printings.

Computer Programs by Chapter and Section

1.0	`flmoon`	calculate phases of the moon by date
1.1	`julday`	Julian Day number from calendar date
1.1	`badluk`	Friday the 13th when the moon is full
1.1	`caldat`	calendar date from Julian day number
2.1	`gaussj`	Gauss-Jordan matrix inversion and linear equation solution
2.3	`ludcmp`	linear equation solution, LU decomposition
2.3	`lubksb`	linear equation solution, backsubstitution
2.4	`tridag`	solution of tridiagonal systems
2.4	`banmul`	multiply vector by band diagonal matrix
2.4	`bandec`	band diagonal systems, decomposition
2.4	`banbks`	band diagonal systems, backsubstitution
2.5	`mprove`	linear equation solution, iterative improvement
2.6	`svbksb`	singular value backsubstitution
2.6	`svdcmp`	singular value decomposition of a matrix
2.6	`pythag`	calculate $(a^2 + b^2)^{1/2}$ without overflow
2.7	`cyclic`	solution of cyclic tridiagonal systems
2.7	`sprsin`	convert matrix to sparse format
2.7	`sprsax`	product of sparse matrix and vector
2.7	`sprstx`	product of transpose sparse matrix and vector
2.7	`sprstp`	transpose of sparse matrix
2.7	`sprspm`	pattern multiply two sparse matrices
2.7	`sprstm`	threshold multiply two sparse matrices
2.7	`linbcg`	biconjugate gradient solution of sparse systems
2.7	`snrm`	used by `linbcg` for vector norm
2.7	`atimes`	used by `linbcg` for sparse multiplication
2.7	`asolve`	used by `linbcg` for preconditioner
2.8	`vander`	solve Vandermonde systems
2.8	`toeplz`	solve Toeplitz systems
2.9	`choldc`	Cholesky decomposition
2.9	`cholsl`	Cholesky backsubstitution
2.10	`qrdcmp`	QR decomposition
2.10	`qrsolv`	QR backsubstitution
2.10	`rsolv`	right triangular backsubstitution
2.10	`qrupdt`	update a QR decomposition
2.10	`rotate`	Jacobi rotation used by `qrupdt`
3.1	`polint`	polynomial interpolation
3.2	`ratint`	rational function interpolation
3.3	`spline`	construct a cubic spline
3.3	`splint`	cubic spline interpolation
3.4	`locate`	search an ordered table by bisection

3.4	hunt	search a table when calls are correlated
3.5	polcoe	polynomial coefficients from table of values
3.5	polcof	polynomial coefficients from table of values
3.6	polin2	two-dimensional polynomial interpolation
3.6	bcucof	construct two-dimensional bicubic
3.6	bcuint	two-dimensional bicubic interpolation
3.6	splie2	construct two-dimensional spline
3.6	splin2	two-dimensional spline interpolation
4.2	trapzd	trapezoidal rule
4.2	qtrap	integrate using trapezoidal rule
4.2	qsimp	integrate using Simpson's rule
4.3	qromb	integrate using Romberg adaptive method
4.4	midpnt	extended midpoint rule
4.4	qromo	integrate using open Romberg adaptive method
4.4	midinf	integrate a function on a semi-infinite interval
4.4	midsql	integrate a function with lower square-root singularity
4.4	midsqu	integrate a function with upper square-root singularity
4.4	midexp	integrate a function that decreases exponentially
4.5	qgaus	integrate a function by Gaussian quadratures
4.5	gauleg	Gauss-Legendre weights and abscissas
4.5	gaulag	Gauss-Laguerre weights and abscissas
4.5	gauher	Gauss-Hermite weights and abscissas
4.5	gaujac	Gauss-Jacobi weights and abscissas
4.5	gaucof	quadrature weights from orthogonal polynomials
4.5	orthog	construct nonclassical orthogonal polynomials
4.6	quad3d	integrate a function over a three-dimensional space
5.1	eulsum	sum a series by Euler–van Wijngaarden algorithm
5.3	ddpoly	evaluate a polynomial and its derivatives
5.3	poldiv	divide one polynomial by another
5.3	ratval	evaluate a rational function
5.7	dfridr	numerical derivative by Ridders' method
5.8	chebft	fit a Chebyshev polynomial to a function
5.8	chebev	Chebyshev polynomial evaluation
5.9	chder	derivative of a function already Chebyshev fitted
5.9	chint	integrate a function already Chebyshev fitted
5.10	chebpc	polynomial coefficients from a Chebyshev fit
5.10	pcshft	polynomial coefficients of a shifted polynomial
5.11	pccheb	inverse of chebpc; use to economize power series
5.12	pade	Padé approximant from power series coefficients
5.13	ratlsq	rational fit by least-squares method
6.1	gammln	logarithm of gamma function
6.1	factrl	factorial function
6.1	bico	binomial coefficients function
6.1	factln	logarithm of factorial function

6.1	beta	beta function
6.2	gammp	incomplete gamma function
6.2	gammq	complement of incomplete gamma function
6.2	gser	series used by gammp and gammq
6.2	gcf	continued fraction used by gammp and gammq
6.2	erf	error function
6.2	erfc	complementary error function
6.2	erfcc	complementary error function, concise routine
6.3	expint	exponential integral E_n
6.3	ei	exponential integral Ei
6.4	betai	incomplete beta function
6.4	betacf	continued fraction used by betai
6.5	bessj0	Bessel function J_0
6.5	bessy0	Bessel function Y_0
6.5	bessj1	Bessel function J_1
6.5	bessy1	Bessel function Y_1
6.5	bessy	Bessel function Y of general integer order
6.5	bessj	Bessel function J of general integer order
6.6	bessi0	modified Bessel function I_0
6.6	bessk0	modified Bessel function K_0
6.6	bessi1	modified Bessel function I_1
6.6	bessk1	modified Bessel function K_1
6.6	bessk	modified Bessel function K of integer order
6.6	bessi	modified Bessel function I of integer order
6.7	bessjy	Bessel functions of fractional order
6.7	beschb	Chebyshev expansion used by bessjy
6.7	bessik	modified Bessel functions of fractional order
6.7	airy	Airy functions
6.7	sphbes	spherical Bessel functions j_n and y_n
6.8	plgndr	Legendre polynomials, associated (spherical harmonics)
6.9	frenel	Fresnel integrals $S(x)$ and $C(x)$
6.9	cisi	cosine and sine integrals Ci and Si
6.10	dawson	Dawson's integral
6.11	rf	Carlson's elliptic integral of the first kind
6.11	rd	Carlson's elliptic integral of the second kind
6.11	rj	Carlson's elliptic integral of the third kind
6.11	rc	Carlson's degenerate elliptic integral
6.11	ellf	Legendre elliptic integral of the first kind
6.11	elle	Legendre elliptic integral of the second kind
6.11	ellpi	Legendre elliptic integral of the third kind
6.11	sncndn	Jacobian elliptic functions
6.12	hypgeo	complex hypergeometric function
6.12	hypser	complex hypergeometric function, series evaluation
6.12	hypdrv	complex hypergeometric function, derivative of
7.1	ran0	random deviate by Park and Miller minimal standard
7.1	ran1	random deviate, minimal standard plus shuffle

7.1	ran2	random deviate by L'Ecuyer long period plus shuffle
7.1	ran3	random deviate by Knuth subtractive method
7.2	expdev	exponential random deviates
7.2	gasdev	normally distributed random deviates
7.3	gamdev	gamma-law distribution random deviates
7.3	poidev	Poisson distributed random deviates
7.3	bnldev	binomial distributed random deviates
7.4	irbit1	random bit sequence
7.4	irbit2	random bit sequence
7.5	psdes	"pseudo-DES" hashing of 64 bits
7.5	ran4	random deviates from DES-like hashing
7.7	sobseq	Sobol's quasi-random sequence
7.8	vegas	adaptive multidimensional Monte Carlo integration
7.8	rebin	sample rebinning used by vegas
7.8	miser	recursive multidimensional Monte Carlo integration
7.8	ranpt	get random point, used by miser
8.1	piksrt	sort an array by straight insertion
8.1	piksr2	sort two arrays by straight insertion
8.1	shell	sort an array by Shell's method
8.2	sort	sort an array by quicksort method
8.2	sort2	sort two arrays by quicksort method
8.3	hpsort	sort an array by heapsort method
8.4	indexx	construct an index for an array
8.4	sort3	sort, use an index to sort 3 or more arrays
8.4	rank	construct a rank table for an array
8.5	select	find the Nth largest in an array
8.5	selip	find the Nth largest, without altering an array
8.5	hpsel	find M largest values, without altering an array
8.6	eclass	determine equivalence classes from list
8.6	eclazz	determine equivalence classes from procedure
9.0	scrsho	graph a function to search for roots
9.1	zbrac	outward search for brackets on roots
9.1	zbrak	inward search for brackets on roots
9.1	rtbis	find root of a function by bisection
9.2	rtflsp	find root of a function by false-position
9.2	rtsec	find root of a function by secant method
9.2	zriddr	find root of a function by Ridders' method
9.3	zbrent	find root of a function by Brent's method
9.4	rtnewt	find root of a function by Newton-Raphson
9.4	rtsafe	find root of a function by Newton-Raphson and bisection
9.5	laguer	find a root of a polynomial by Laguerre's method
9.5	zroots	roots of a polynomial by Laguerre's method with deflation
9.5	zrhqr	roots of a polynomial by eigenvalue methods
9.5	qroot	complex or double root of a polynomial, Bairstow

9.6	mnewt	Newton's method for systems of equations
9.7	lnsrch	search along a line, used by newt
9.7	newt	globally convergent multi-dimensional Newton's method
9.7	fdjac	finite-difference Jacobian, used by newt
9.7	fmin	norm of a vector function, used by newt
9.7	broydn	secant method for systems of equations
10.1	mnbrak	bracket the minimum of a function
10.1	golden	find minimum of a function by golden section search
10.2	brent	find minimum of a function by Brent's method
10.3	dbrent	find minimum of a function using derivative information
10.4	amoeba	minimize in N-dimensions by downhill simplex method
10.4	amotry	evaluate a trial point, used by amoeba
10.5	powell	minimize in N-dimensions by Powell's method
10.5	linmin	minimum of a function along a ray in N-dimensions
10.5	f1dim	function used by linmin
10.6	frprmn	minimize in N-dimensions by conjugate gradient
10.6	df1dim	alternative function used by linmin
10.7	dfpmin	minimize in N-dimensions by variable metric method
10.8	simplx	linear programming maximization of a linear function
10.8	simp1	linear programming, used by simplx
10.8	simp2	linear programming, used by simplx
10.8	simp3	linear programming, used by simplx
10.9	anneal	traveling salesman problem by simulated annealing
10.9	revcst	cost of a reversal, used by anneal
10.9	revers	do a reversal, used by anneal
10.9	trncst	cost of a transposition, used by anneal
10.9	trnspt	do a transposition, used by anneal
10.9	metrop	Metropolis algorithm, used by anneal
10.9	amebsa	simulated annealing in continuous spaces
10.9	amotsa	evaluate a trial point, used by amebsa
11.1	jacobi	eigenvalues and eigenvectors of a symmetric matrix
11.1	eigsrt	eigenvectors, sorts into order by eigenvalue
11.2	tred2	Householder reduction of a real, symmetric matrix
11.3	tqli	eigensolution of a symmetric tridiagonal matrix
11.5	balanc	balance a nonsymmetric matrix
11.5	elmhes	reduce a general matrix to Hessenberg form
11.6	hqr	eigenvalues of a Hessenberg matrix
12.2	four1	fast Fourier transform (FFT) in one dimension
12.3	twofft	fast Fourier transform of two real functions
12.3	realft	fast Fourier transform of a single real function
12.3	sinft	fast sine transform
12.3	cosft1	fast cosine transform with endpoints
12.3	cosft2	"staggered" fast cosine transform
12.4	fourn	fast Fourier transform in multidimensions

12.5	`rlft3`	FFT of real data in two or three dimensions
12.6	`fourfs`	FFT for huge data sets on external media
12.6	`fourew`	rewind and permute files, used by `fourfs`
13.1	`convlv`	convolution or deconvolution of data using FFT
13.2	`correl`	correlation or autocorrelation of data using FFT
13.4	`spctrm`	power spectrum estimation using FFT
13.6	`memcof`	evaluate maximum entropy (MEM) coefficients
13.6	`fixrts`	reflect roots of a polynomial into unit circle
13.6	`predic`	linear prediction using MEM coefficients
13.7	`evlmem`	power spectral estimation from MEM coefficients
13.8	`period`	power spectrum of unevenly sampled data
13.8	`fasper`	power spectrum of unevenly sampled larger data sets
13.8	`spread`	extirpolate value into array, used by `fasper`
13.9	`dftcor`	compute endpoint corrections for Fourier integrals
13.9	`dftint`	high-accuracy Fourier integrals
13.10	`wt1`	one-dimensional discrete wavelet transform
13.10	`daub4`	Daubechies 4-coefficient wavelet filter
13.10	`pwtset`	initialize coefficients for `pwt`
13.10	`pwt`	partial wavelet transform
13.10	`wtn`	multidimensional discrete wavelet transform
14.1	`moment`	calculate moments of a data set
14.2	`ttest`	Student's t-test for difference of means
14.2	`avevar`	calculate mean and variance of a data set
14.2	`tutest`	Student's t-test for means, case of unequal variances
14.2	`tptest`	Student's t-test for means, case of paired data
14.2	`ftest`	F-test for difference of variances
14.3	`chsone`	chi-square test for difference between data and model
14.3	`chstwo`	chi-square test for difference between two data sets
14.3	`ksone`	Kolmogorov-Smirnov test of data against model
14.3	`kstwo`	Kolmogorov-Smirnov test between two data sets
14.3	`probks`	Kolmogorov-Smirnov probability function
14.4	`cntab1`	contingency table analysis using chi-square
14.4	`cntab2`	contingency table analysis using entropy measure
14.5	`pearsn`	Pearson's correlation between two data sets
14.6	`spear`	Spearman's rank correlation between two data sets
14.6	`crank`	replaces array elements by their rank
14.6	`kendl1`	correlation between two data sets, Kendall's tau
14.6	`kendl2`	contingency table analysis using Kendall's tau
14.7	`ks2d1s`	K–S test in two dimensions, data vs. model
14.7	`quadct`	count points by quadrants, used by `ks2d1s`
14.7	`quadvl`	quadrant probabilities, used by `ks2d1s`
14.7	`ks2d2s`	K–S test in two dimensions, data vs. data
14.8	`savgol`	Savitzky-Golay smoothing coefficients
15.2	`fit`	least-squares fit data to a straight line

15.3	fitexy	fit data to a straight line, errors in both x and y
15.3	chixy	used by fitexy to calculate a χ^2
15.4	lfit	general linear least-squares fit by normal equations
15.4	covsrt	rearrange covariance matrix, used by lfit
15.4	svdfit	linear least-squares fit by singular value decomposition
15.4	svdvar	variances from singular value decomposition
15.4	fpoly	fit a polynomial using lfit or svdfit
15.4	fleg	fit a Legendre polynomial using lfit or svdfit
15.5	mrqmin	nonlinear least-squares fit, Marquardt's method
15.5	mrqcof	used by mrqmin to evaluate coefficients
15.5	fgauss	fit a sum of Gaussians using mrqmin
15.7	medfit	fit data to a straight line robustly, least absolute deviation
15.7	rofunc	fit data robustly, used by medfit
16.1	rk4	integrate one step of ODEs, fourth-order Runge-Kutta
16.1	rkdumb	integrate ODEs by fourth-order Runge-Kutta
16.2	rkqs	integrate one step of ODEs with accuracy monitoring
16.2	rkck	Cash-Karp-Runge-Kutta step used by rkqs
16.2	odeint	integrate ODEs with accuracy monitoring
16.3	mmid	integrate ODEs by modified midpoint method
16.4	bsstep	integrate ODEs, Bulirsch-Stoer step
16.4	pzextr	polynomial extrapolation, used by bsstep
16.4	rzextr	rational function extrapolation, used by bsstep
16.5	stoerm	integrate conservative second-order ODEs
16.6	stiff	integrate stiff ODEs by fourth-order Rosenbrock
16.6	jacobn	sample Jacobian routine for stiff
16.6	derivs	sample derivatives routine for stiff
16.6	simpr	integrate stiff ODEs by semi-implicit midpoint rule
16.6	stifbs	integrate stiff ODEs, Bulirsch-Stoer step
17.1	shoot	solve two point boundary value problem by shooting
17.2	shootf	ditto, by shooting to a fitting point
17.3	solvde	two point boundary value problem, solve by relaxation
17.3	bksub	backsubstitution, used by solvde
17.3	pinvs	diagonalize a sub-block, used by solvde
17.3	red	reduce columns of a matrix, used by solvde
17.4	sfroid	spheroidal functions by method of solvde
17.4	difeq	spheroidal matrix coefficients, used by sfroid
17.4	sphoot	spheroidal functions by method of shoot
17.4	sphfpt	spheroidal functions by method of shootf
18.1	fred2	solve linear Fredholm equations of the second kind
18.1	fredin	interpolate solutions obtained with fred2
18.2	voltra	linear Volterra equations of the second kind
18.3	wwghts	quadrature weights for an arbitrarily singular kernel
18.3	kermom	sample routine for moments of a singular kernel
18.3	quadmx	sample routine for a quadrature matrix

18.3	`fredex`	example of solving a singular Fredholm equation
19.5	`sor`	elliptic PDE solved by successive overrelaxation method
19.6	`mglin`	linear elliptic PDE solved by multigrid method
19.6	`rstrct`	half-weighting restriction, used by `mglin`, `mgfas`
19.6	`interp`	bilinear prolongation, used by `mglin`, `mgfas`
19.6	`addint`	interpolate and add, used by `mglin`
19.6	`slvsml`	solve on coarsest grid, used by `mglin`
19.6	`relax`	Gauss-Seidel relaxation, used by `mglin`
19.6	`resid`	calculate residual, used by `mglin`
19.6	`copy`	utility used by `mglin`, `mgfas`
19.6	`fill0`	utility used by `mglin`
19.6	`maloc`	memory allocation utility used by `mglin`, `mgfas`
19.6	`mgfas`	nonlinear elliptic PDE solved by multigrid method
19.6	`relax2`	Gauss-Seidel relaxation, used by `mgfas`
19.6	`slvsm2`	solve on coarsest grid, used by `mgfas`
19.6	`lop`	applies nonlinear operator, used by `mgfas`
19.6	`matadd`	utility used by `mgfas`
19.6	`matsub`	utility used by `mgfas`
19.6	`anorm2`	utility used by `mgfas`
20.1	`machar`	diagnose computer's floating arithmetic
20.2	`igray`	Gray code and its inverse
20.3	`icrc1`	cyclic redundancy checksum, used by `icrc`
20.3	`icrc`	cyclic redundancy checksum
20.3	`decchk`	decimal check digit calculation or verification
20.4	`hufmak`	construct a Huffman code
20.4	`hufapp`	append bits to a Huffman code, used by `hufmak`
20.4	`hufenc`	use Huffman code to encode and compress a character
20.4	`hufdec`	use Huffman code to decode and decompress a character
20.5	`arcmak`	construct an arithmetic code
20.5	`arcode`	encode or decode a character using arithmetic coding
20.5	`arcsum`	add integer to byte string, used by `arcode`
20.6	`mpops`	multiple precision arithmetic, simpler operations
20.6	`mpmul`	multiple precision multiply, using FFT methods
20.6	`mpinv`	multiple precision reciprocal
20.6	`mpdiv`	multiple precision divide and remainder
20.6	`mpsqrt`	multiple precision square root
20.6	`mp2dfr`	multiple precision conversion to decimal base
20.6	`mppi`	multiple precision example, compute many digits of π

Chapter 1. Preliminaries

1.0 Introduction

This book, like its predecessor edition, is supposed to teach you methods of numerical computing that are practical, efficient, and (insofar as possible) elegant. We presume throughout this book that you, the reader, have particular tasks that you want to get done. We view our job as educating you on how to proceed. Occasionally we may try to reroute you briefly onto a particularly beautiful side road; but by and large, we will guide you along main highways that lead to practical destinations.

Throughout this book, you will find us fearlessly editorializing, telling you what you should and shouldn't do. This prescriptive tone results from a conscious decision on our part, and we hope that you will not find it irritating. We do not claim that our advice is infallible! Rather, we are reacting against a tendency, in the textbook literature of computation, to discuss every possible method that has ever been invented, without ever offering a practical judgment on relative merit. We do, therefore, offer you our practical judgments whenever we can. As you gain experience, you will form your own opinion of how reliable our advice is.

We presume that you are able to read computer programs in `FORTRAN`, that being the language of this version of *Numerical Recipes* (Second Edition). The book *Numerical Recipes in C* (Second Edition) is separately available, if you prefer to program in that language. Earlier editions of *Numerical Recipes in Pascal* and *Numerical Recipes Routines and Examples in BASIC* are also available; while not containing the additional material of the Second Edition versions in `C` and `FORTRAN`, these versions are perfectly serviceable if `Pascal` or `BASIC` is your language of choice.

When we include programs in the text, they look like this:

```
SUBROUTINE flmoon(n,nph,jd,frac)
INTEGER jd,n,nph
REAL frac,RAD
PARAMETER (RAD=3.14159265/180.)
```

Our programs begin with an introductory comment summarizing their purpose and explaining their calling sequence. This routine calculates the phases of the moon. Given an integer `n` and a code `nph` for the phase desired (`nph` $= 0$ for new moon, 1 for first quarter, 2 for full, 3 for last quarter), the routine returns the Julian Day Number `jd`, and the fractional part of a day `frac` to be added to it, of the `n`th such phase since January, 1900. Greenwich Mean Time is assumed.

```
INTEGER i
REAL am,as,c,t,t2,xtra
c=n+nph/4.                              This is how we comment an individual line.
t=c/1236.85
t2=t**2
```

```
as=359.2242+29.105356*c                 You aren't really intended to understand this al-
am=306.0253+385.816918*c+0.010730*t2        gorithm, but it does work!
jd=2415020+28*n+7*nph
xtra=0.75933+1.53058868*c+(1.178e-4-1.55e-7*t)*t2
if(nph.eq.0.or.nph.eq.2)then
    xtra=xtra+(0.1734-3.93e-4*t)*sin(RAD*as)-0.4068*sin(RAD*am)
else if(nph.eq.1.or.nph.eq.3)then
    xtra=xtra+(0.1721-4.e-4*t)*sin(RAD*as)-0.6280*sin(RAD*am)
else
    pause 'nph is unknown in flmoon'    This is how we will indicate error conditions.
endif
if(xtra.ge.0.)then
    i=int(xtra)
else
    i=int(xtra-1.)
endif
jd=jd+i
frac=xtra-i
return
END
```

A few remarks about our typographical conventions and programming style are in order at this point:

- It is good programming practice to declare all variables and identifiers in explicit "type" statements (`REAL`, `INTEGER`, etc.), even though the implicit declaration rules of `FORTRAN` do not require this. We will always do so. (As an aside to non-`FORTRAN` programmers, the implicit declaration rules are that variables which begin with the letters `i,j,k,l,m,n` are implicitly declared to be type `INTEGER`, while all other variables are implicitly declared to be type `REAL`. Explicit declarations override these conventions.)
- In sympathy with modular and object-oriented programming practice, we separate, typographically, a routine's "public" or "interface" section from its "private" or "implementation" section. We do this even though `FORTRAN` is by no means a modular or object-oriented language: the separation makes sense simply as good programming style.
- The *public* section contains the calling interface and declarations of its variables. We find it useful to consider `PARAMETER` statements, and their associated declarations, as also being in the public section, since a user may want to modify parameter values to suit a particular purpose. `COMMON` blocks are likewise usually part of the public section, since they involve communication between routines.
- As the last entry in the public section, we will, where applicable, put a standardized comment line with the word `USES` (not a `FORTRAN` keyword), followed by a list of all external subroutines and functions that the routine references, excluding built-in `FORTRAN` functions. (For examples, see the routines in §6.1.)
- An introductory comment, set in type as an indented paragraph, separates the public section from the private or implementation section.
- Within the introductory comments, as well as in the text, we will frequently use the notation `a(1:m)` to mean "the array elements `a(1)`, `a(2)`, ..., `a(m)`." Likewise, notations like `b(2:7)` or `c(1:m,1:n)` are to be

interpreted as ranges of array indices. (This use of colon to denote ranges comes from FORTRAN-77's syntax for array declarators and character substrings.)

- The *implementation section* contains the declarations of variables that are used only internally in the routine, any necessary SAVE statements for static variables (variables that must be preserved between calls to the routine), and of course the routine's actual executable code.
- Case is not significant in FORTRAN, so it can be used to promote readability. Our convention is to use upper case for two different, nonconflicting, purposes. First, nonexecutable compiler keywords are in upper case (e.g., SUBROUTINE, REAL, COMMON); second, parameter identifiers are in upper case. The reason for capitalizing parameters is that, because their values are liable to be modified, the user often needs to scan the implementation section of code to see exactly how the parameters are used.
- For simplicity, we adopt the convention of handling all errors and exceptional cases by the pause statement. In general, we do not intend that you continue program execution after a pause occurs, but FORTRAN allows you to do so — if you want to see what kind of wrong answer or catastrophic error results. In many applications, you will want to modify our programs to do more sophisticated error handling, for example to return with an error flag set, or call an error-handling routine.
- In the printed form of this book, we take some special typographical liberties regarding statement labels, and do ... continue constructions. These are described in §1.1. Note that no such liberties are taken in the machine-readable *Numerical Recipes* diskettes, where all routines are in standard ANSI FORTRAN-77.

Computational Environment and Program Validation

Our goal is that the programs in this book be as portable as possible, across different platforms (models of computer), across different operating systems, and across different FORTRAN compilers. As surrogates for the large number of possible combinations, we have tested all the programs in this book on the combinations of machines, operating systems, and compilers shown on the accompanying table. More generally, the programs should run without modification on any compiler that implements the ANSI FORTRAN-77 standard. At the time of writing, there are not enough installed implementations of the successor FORTRAN-90 standard to justify our using any of its more advanced features. Since FORTRAN-90 is backwards-compatible with FORTRAN-77, there should be no difficulty in using the programs in this book on FORTRAN-90 compilers, as they become available.

In validating the programs, we have taken the program source code directly from the machine-readable form of the book's manuscript, to decrease the chance of propagating typographical errors. "Driver" or demonstration programs that we used as part of our validations are available separately as the *Numerical Recipes Example Book (FORTRAN)*, as well as in machine-readable form. If you plan to use more than a few of the programs in this book, or if you plan to use programs in this book on more than one different computer, then you may find it useful to obtain a copy of these demonstration programs.

Tested Machines and Compilers		
Hardware	O/S Version	Compiler Version
IBM PC compatible 486/33	MS-DOS 5.0	Microsoft Fortran 5.1
IBM RS6000	AIX 3.0	IBM AIX XL FORTRAN Compiler/6000
IBM PC-RT	BSD UNIX 4.3	"UNIX Fortran 77"
DEC VAX 4000	VMS 5.4	VAX Fortran 5.4
DEC VAXstation 2000	BSD UNIX 4.3	Berkeley f77 2.0 (4.3 bsd, SCCS lev. 6)
DECstation 5000/200	ULTRIX 4.2	DEC Fortran for ULTRIX RISC 3.1
DECsystem 5400	ULTRIX 4.1	MIPS f77 2.10
Sun SPARCstation 2	SunOS 4.1	Sun Fortran 1.4 (SC 1.0)
Apple Macintosh	System 6.0.7 / MPW 3.2	Absoft Fortran 77 Compiler 3.1.2

Of course we would be foolish to claim that there are no bugs in our programs, and we do not make such a claim. We have been very careful, and have benefitted from the experience of the many readers who have written to us. If you find a new bug, please document it and tell us!

Compatibility with the First Edition

If you are accustomed to the *Numerical Recipes* routines of the First Edition, rest assured: almost all of them are still here, with the same names and functionalities, often with major improvements in the code itself. In addition, we hope that you will soon become equally familiar with the added capabilities of the more than 100 routines that are new to this edition.

We have retired a small number of First Edition routines, those that we believe to be clearly dominated by better methods implemented in this edition. A table, following, lists the retired routines and suggests replacements.

First Edition users should also be aware that some routines common to both editions have alterations in their calling interfaces, so are not directly "plug compatible." A fairly complete list is: `chsone`, `chstwo`, `covsrt`, `dfpmin`, `laguer`, `lfit`, `memcof`, `mrqcof`, `mrqmin`, `pzextr`, `ran4`, `realft`, `rzextr`, `shoot`, `shootf`. There may be others (depending in part on which printing of the First Edition is taken for the comparison). If you have written software of any appreciable complexity that is dependent on First Edition routines, we do *not* recommend blindly replacing them by the corresponding routines in this book. We do recommend that any new programming efforts use the new routines.

About References

You will find references, and suggestions for further reading, listed at the end of most sections of this book. References are cited in the text by bracketed numbers like this [1].

Because computer algorithms often circulate informally for quite some time before appearing in a published form, the task of uncovering "primary literature"

Previous Routines Omitted from This Edition		
Name(s)	Replacement(s)	Comment
ADI	mglin or mgfas	better method
COSFT	cosft1 or cosft2	choice of boundary conditions
CEL, EL2	rf, rd, rj, rc	better algorithms
DES, DESKS	ran4 now uses psdes	was too slow
MDIAN1, MDIAN2	select, selip	more general
QCKSRT	sort	name change (SORT is now hpsort)
RKQC	rkqs	better method
SMOOFT	use convlv with coefficients from savgol	
SPARSE	linbcg	more general

is sometimes quite difficult. We have not attempted this, and we do not pretend to any degree of bibliographical completeness in this book. For topics where a substantial secondary literature exists (discussion in textbooks, reviews, etc.) we have consciously limited our references to a few of the more useful secondary sources, especially those with good references to the primary literature. Where the existing secondary literature is insufficient, we give references to a few primary sources that are intended to serve as starting points for further reading, not as complete bibliographies for the field.

The order in which references are listed is not necessarily significant. It reflects a compromise between listing cited references in the order cited, and listing suggestions for further reading in a roughly prioritized order, with the most useful ones first.

The remaining two sections of this chapter review some basic concepts of programming (control structures, etc.) and of numerical analysis (roundoff error, etc.). Thereafter, we plunge into the substantive material of the book.

CITED REFERENCES AND FURTHER READING:

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [1]

1.1 Program Organization and Control Structures

We sometimes like to point out the close analogies between computer programs, on the one hand, and written poetry or written musical scores, on the other. All three present themselves as visual media, symbols on a two-dimensional page or computer screen. Yet, in all three cases, the visual, two-dimensional, *frozen-in-time* representation communicates (or is supposed to communicate) something rather

Previous Routines Omitted from This Edition		
Name(s)	Replacement(s)	Comment
ADI	mglin or mgfas	better method
COSFT	cosft1 or cosft2	choice of boundary conditions
CEL, EL2	rf, rd, rj, rc	better algorithms
DES, DESKS	ran4 now uses psdes	was too slow
MDIAN1, MDIAN2	select, selip	more general
QCKSRT	sort	name change (SORT is now hpsort)
RKQC	rkqs	better method
SMOOFT	use convlv with coefficients from savgol	
SPARSE	linbcg	more general

is sometimes quite difficult. We have not attempted this, and we do not pretend to any degree of bibliographical completeness in this book. For topics where a substantial secondary literature exists (discussion in textbooks, reviews, etc.) we have consciously limited our references to a few of the more useful secondary sources, especially those with good references to the primary literature. Where the existing secondary literature is insufficient, we give references to a few primary sources that are intended to serve as starting points for further reading, not as complete bibliographies for the field.

The order in which references are listed is not necessarily significant. It reflects a compromise between listing cited references in the order cited, and listing suggestions for further reading in a roughly prioritized order, with the most useful ones first.

The remaining two sections of this chapter review some basic concepts of programming (control structures, etc.) and of numerical analysis (roundoff error, etc.). Thereafter, we plunge into the substantive material of the book.

CITED REFERENCES AND FURTHER READING:

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [1]

1.1 Program Organization and Control Structures

We sometimes like to point out the close analogies between computer programs, on the one hand, and written poetry or written musical scores, on the other. All three present themselves as visual media, symbols on a two-dimensional page or computer screen. Yet, in all three cases, the visual, two-dimensional, *frozen-in-time* representation communicates (or is supposed to communicate) something rather

different, namely a process that *unfolds in time*. A poem is meant to be read; music, played; a program, executed as a sequential series of computer instructions.

In all three cases, the target of the communication, in its visual form, is a human being. The goal is to transfer to him/her, as efficiently as can be accomplished, the greatest degree of understanding, in advance, of how the process *will* unfold in time. In poetry, this human target is the reader. In music, it is the performer. In programming, it is the program user.

Now, you may object that the target of communication of a program is not a human but a computer, that the program user is only an irrelevant intermediary, a lackey who feeds the machine. This is perhaps the case in the situation where the business executive pops a diskette into a desktop computer and feeds that computer a black-box program in binary executable form. The computer, in this case, doesn't much care whether that program was written with "good programming practice" or not.

We envision, however, that you, the readers of this book, are in quite a different situation. You need, or want, to know not just *what* a program does, but also *how* it does it, so that you can tinker with it and modify it to your particular application. You need others to be able to see what you have done, so that they can criticize or admire. In such cases, where the desired goal is *maintainable* or *reusable* code, the targets of a program's communication are surely human, not machine.

One key to achieving good programming practice is to recognize that programming, music, and poetry — all three being symbolic constructs of the human brain — are naturally structured into hierarchies that have many different nested levels. Sounds (phonemes) form small meaningful units (morphemes) which in turn form words; words group into phrases, which group into sentences; sentences make paragraphs, and these are organized into higher levels of meaning. Notes form musical phrases, which form themes, counterpoints, harmonies, etc.; which form movements, which form concertos, symphonies, and so on.

The structure in programs is equally hierarchical. Appropriately, good programming practice brings different techniques to bear on the different levels [1-3]. At a low level is the `ascii` character set. Then, constants, identifiers, operands, operators. Then program statements, like `a(j+1)=b+c/3.0`. Here, the best programming advice is simply *be clear*, or (correspondingly) *don't be too tricky*. You might momentarily be proud of yourself at writing the single line

```
k=(2-j)*(1+3*j)/2
```

if you want to permute cyclically one of the values $\mathtt{j} = (0, 1, 2)$ into respectively $\mathtt{k} = (1, 2, 0)$. You will regret it later, however, when you try to understand that line. Better, and likely also faster, is

```
k=j+1
if (k.eq.3) k=0
```

Many programming stylists would even argue for the ploddingly literal

```
if (j.eq.0) then
    k=1
else if (j.eq.1) then
    k=2
```

```
else if (j.eq.2) then
    k=0
else
    pause 'never get here'
endif
```

on the grounds that it is both clear and additionally safeguarded from wrong assumptions about the possible values of `j`. Our preference among the implementations is for the middle one.

In this simple example, we have in fact traversed several levels of hierarchy: Statements frequently come in "groups" or "blocks" which make sense only taken as a whole. The middle fragment above is one example. Another is

```
swap=a(j)
a(j)=b(j)
b(j)=swap
```

which makes immediate sense to any programmer as the exchange of two variables, while

```
sum=0.0
ans=0.0
n=1
```

is very likely to be an initialization of variables prior to some iterative process. This level of hierarchy in a program is usually evident to the eye. It is good programming practice to put in comments at this level, e.g., "initialize" or "exchange variables."

The next level is that of *control structures*. These are things like the `if...then...else` clauses in the example above, `do` loops, and so on. This level is sufficiently important, and relevant to the hierarchical level of the routines in this book, that we will come back to it just below.

At still higher levels in the hierarchy, we have (in `FORTRAN`) subroutines, functions, and the whole "global" organization of the computational task to be done. In the musical analogy, we are now at the level of movements and complete works. At these levels, *modularization* and *encapsulation* become important programming concepts, the general idea being that program units should interact with one another only through clearly defined and narrowly circumscribed interfaces. Good modularization practice is an essential prerequisite to the success of large, complicated software projects, especially those employing the efforts of more than one programmer. It is also good practice (if not quite as essential) in the less massive programming tasks that an individual scientist, or reader of this book, encounters.

Some computer languages, such as Modula-2 and `C++`, promote good modularization with higher-level language constructs, absent in `FORTRAN`-77. In Modula-2, for example, subroutines, type definitions, and data structures can be encapsulated into "modules" that communicate through declared public interfaces and whose internal workings are hidden from the rest of the program [4]. In the `C++` language, the key concept is "class," a user-definable generalization of data type that provides for data hiding, automatic initialization of data, memory management, dynamic typing, and operator overloading (i.e., the user-definable extension of operators like `+` and `*` so as to be appropriate to operands in any particular class) [5]. Properly used in defining the data structures that are passed between program units, classes

can clarify and circumscribe these units' public interfaces, reducing the chances of programming error and also allowing a considerable degree of compile-time and run-time error checking.

Beyond modularization, though depending on it, lie the concepts of *object-oriented programming*. Here a programming language, such as C++ or Turbo Pascal 5.5 [6], allows a module's public interface to accept redefinitions of types or actions, and these redefinitions become shared all the way down through the module's hierarchy (so-called *polymorphism*). For example, a routine written to invert a matrix of real numbers could — dynamically, at run time — be made able to handle complex numbers by overloading complex data types and corresponding definitions of the arithmetic operations. Additional concepts of *inheritance* (the ability to define a data type that "inherits" all the structure of another type, plus additional structure of its own), and *object extensibility* (the ability to add functionality to a module without access to its source code, e.g., at run time), also come into play.

We have not attempted to modularize, or make objects out of, the routines in this book, for at least two reasons. First, the chosen language, FORTRAN-77, does not really make this possible. Second, we envision that you, the reader, might want to incorporate the algorithms in this book, a few at a time, into modules or objects with a structure of your own choosing. There does not exist, at present, a standard or accepted set of "classes" for scientific object-oriented computing. While we might have tried to invent such a set, doing so would have inevitably tied the algorithmic content of the book (which is its *raison d'être*) to some rather specific, and perhaps haphazard, set of choices regarding class definitions.

On the other hand, we are not unfriendly to the goals of modular and object-oriented programming. Within the limits of FORTRAN, we have therefore tried to structure our programs to be "object friendly," principally via the clear delineation of interface vs. implementation (§1.0) and the explicit declaration of variables. Within our implementation sections, we have paid particular attention to the practices of *structured programming*, as we now discuss.

Control Structures

An executing program unfolds in time, but not strictly in the linear order in which the statements are written. Program statements that affect the order in which statements are executed, or that affect whether statements are executed, are called *control statements*. Control statements never make useful sense by themselves. They make sense only in the context of the groups or blocks of statements that they in turn control. If you think of those blocks as paragraphs containing sentences, then the control statements are perhaps best thought of as the indentation of the paragraph and the punctuation between the sentences, not the words within the sentences.

We can now say what the goal of structured programming is. It is *to make program control manifestly apparent in the visual presentation of the program.* You see that this goal has nothing at all to do with how the computer sees the program. As already remarked, computers don't care whether you use structured programming or not. Human readers, however, *do* care. You yourself will also care, once you discover how much easier it is to perfect and debug a well-structured program than one whose control structure is obscure.

You accomplish the goals of structured programming in two complementary ways. First, you acquaint yourself with the small number of essential control structures that occur over and over again in programming, and that are therefore given convenient representations in most programming languages. You should learn to think about your programming tasks, insofar as possible, exclusively in terms of these standard control structures. In writing programs, you should get into the habit of representing these standard control structures in consistent, conventional ways.

"Doesn't this inhibit *creativity*?" our students sometimes ask. Yes, just as Mozart's creativity was inhibited by the sonata form, or Shakespeare's by the metrical requirements of the sonnet. The point is that creativity, when it is meant to communicate, does *well* under the inhibitions of appropriate restrictions on format.

Second, you *avoid*, insofar as possible, control statements whose controlled blocks or objects are difficult to discern at a glance. This means, in practice, that *you must try to avoid statement labels and* `goto`*'s.* It is not the `goto`'s that are dangerous (although they do interrupt one's reading of a program); the statement labels are the hazard. In fact, whenever you encounter a statement label while reading a program, you will soon become conditioned to get a sinking feeling in the pit of your stomach. Why? Because the following questions will, by habit, immediately spring to mind: Where did control come *from* in a branch to this label? It could be anywhere in the routine! What circumstances resulted in a branch to this label? They could be anything! Certainty becomes uncertainty, understanding dissolves into a morass of possibilities.

Some older languages, notably 1966 `FORTRAN` and to a lesser extent `FORTRAN`-77, *require* statement labels in the construction of certain standard control structures. We will see this in more detail below. This is a demerit for these languages. In such cases, you must use labels as required. But you should never branch to them independently of the standard control structure. If you must branch, let it be to an additional label, one that is not masquerading as part of a standard control structure.

We call labels that are part of a standard construction and never otherwise branched to *tame labels.* They do not interfere with structured programming in any way, except possibly typographically as distractions to the eye.

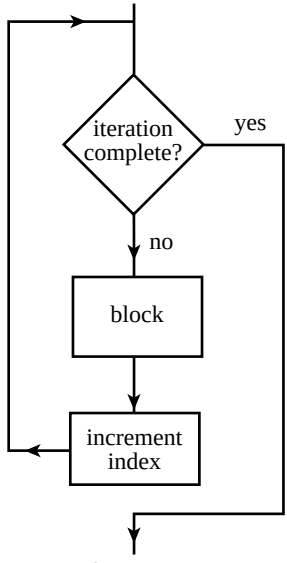
Some examples are now in order to make these considerations more concrete (see Figure 1.1.1).

Catalog of Standard Structures

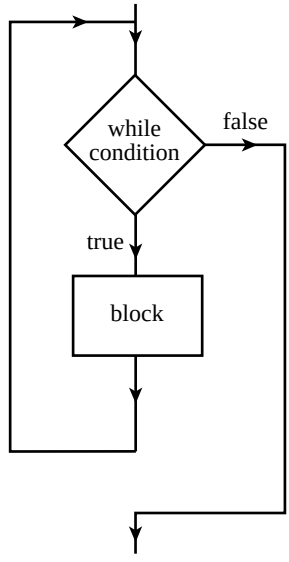
Iteration. In `FORTRAN`, simple iteration is performed with a `do` loop, for example

```
      do 10 j=2,1000
          b(j)=a(j-1)
          a(j-1)=j
10    continue
```

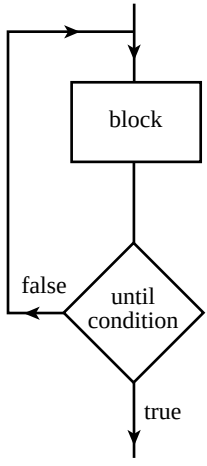
Notice how we always indent the block of code that is acted upon by the control structure, leaving the structure itself unindented. The statement label `10` in this example is a tame label. The majority of modern implementations of `FORTRAN`-77 provide a nonstandard language extension that obviates the tame label. Originally



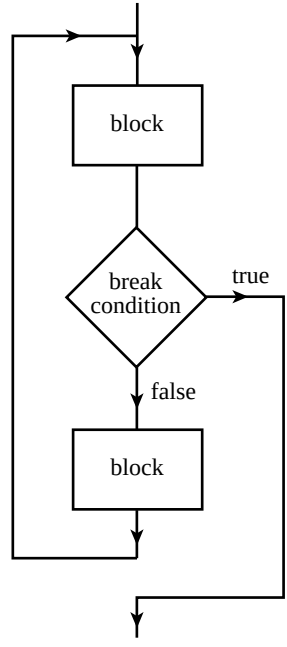
DO iteration
(a)



DO WHILE iteration
(b)

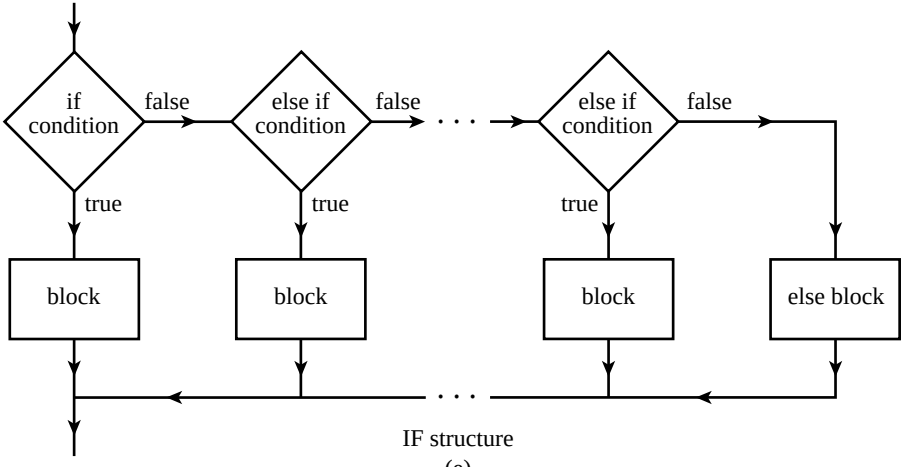


DO UNTIL iteration
(c)

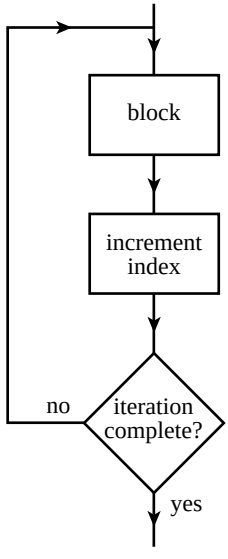


BREAK iteration
(d)

Figure 1.1.1. Standard control structures used in structured programming: (a) DO iteration; (b) DO WHILE iteration; (c) DO UNTIL iteration; (d) BREAK iteration; (e) IF structure; (f) obsolete form of DO iteration found in `FORTRAN-66`, where the block is executed once even if the iteration condition is initially not satisfied.



IF structure
(e)



FORTRAN-66 DO (obsolete)
(f)

Figure 1.1.1. Standard control structures used in structured programming (see caption on previous page).

introduced in Digital Equipment Corporations's VAX-11 FORTRAN, the "enddo" statement is used as

```
do j=2,1000
    b(j)=a(j-1)
    a(j-1)=j
enddo
```

In fact, it was a terrible mistake that the American National Standard for FORTRAN-77 (ANSI X3.9–1978) failed to provide an enddo or equivalent construction. This mistake by the people who write standards, whoever they are, presents us now, more than 15 years later, with a painful quandary: Do we stick to the standard, and clutter our programs with tame labels? Or do we adopt a nonstandard (albeit widely implemented) FORTRAN construction like enddo?

We have adopted a compromise position. Standards, even imperfect standards, are terribly important and highly necessary in a time of rapid evolution in computers and their applications. Therefore, all machine-readable forms of our programs (e.g., the diskettes that you can order from the publisher — see back of this book) are strictly FORTRAN-77 compliant. (Well, *almost* strictly: there is a minor anomaly regarding bit manipulation functions, see below.) In particular, do blocks always end with labeled continue statements, as in the first example above.

In the printed version of this book, however, we make use of typography to mitigate the standard's deficiencies. The statement label that follows the do is printed in small type — as a signal that it is a tame label that you can safely ignore. And, the word "continue" is printed as "enddo", which you may regard as a *very* peculiar change of font! The example above, in our adopted typographical format, is

```
do 10 j=2,1000
    b(j)=a(j-1)
    a(j-1)=j
enddo 10
```

(Notice that we also take the typographical liberty of writing the tame label after the "continue" statement, rather than before.)

A nested do loop looks like this:

```
do 12 j=1,20
    s(j)=0.
    do 11 k=5,10
        s(j)=s(j)+a(j,k)
    enddo 11
enddo 12
```

Generally, the numerical values of the tame labels are chosen to put the enddo's (labeled continue's on the diskette) into ascending numerical order, hence the do 12 before the do 11 in the above example.

IF structure. In this structure the FORTRAN-77 standard is exemplary. Here is a working program that consists dominantly of if control statements:

```
FUNCTION julday(mm,id,iyyy)
INTEGER julday,id,iyyy,mm,IGREG
PARAMETER (IGREG=15+31*(10+12*1582))        Gregorian Calendar adopted Oct. 15, 1582.
    In this routine julday returns the Julian Day Number that begins at noon of the calendar
    date specified by month mm, day id, and year iyyy, all integer variables. Positive year
    signifies A.D.; negative, B.C. Remember that the year after 1 B.C. was 1 A.D.
INTEGER ja,jm,jy
jy=iyyy
if (jy.eq.0) pause 'julday: there is no year zero'
if (jy.lt.0) jy=jy+1
if (mm.gt.2) then                           Here is an example of a block IF-structure.
    jm=mm+1
else
    jy=jy-1
    jm=mm+13
endif
julday=int(365.25*jy)+int(30.6001*jm)+id+1720995
if (id+31*(mm+12*iyyy).ge.IGREG) then       Test whether to change to Gregorian Calen-
    ja=int(0.01*jy)                             dar.
    julday=julday+2-ja+int(0.25*ja)
endif
return
END
```

(Astronomers number each 24-hour period, starting and ending at *noon*, with a unique integer, the Julian Day Number [7]. Julian Day Zero was a very long time ago; a convenient reference point is that Julian Day 2440000 began at noon of May 23, 1968. If you know the Julian Day Number that begins at noon of a given calendar date, then the day of the week of that date is obtained by adding 1 and taking the result modulo base 7; a zero answer corresponds to Sunday, 1 to Monday, ..., 6 to Saturday.)

Do-While iteration. Most good languages, except FORTRAN, provide for structures like the following C example:

```
while (n<1000) {
    n=2*n;
    j++;                    In C this has the meaning j=j+1.
}
```

In fact, many FORTRAN implementations have the nonstandard extension

```
do while (n.lt.1000)
    n=2*n
    j=j+1
enddo
```

Within the FORTRAN-77 standard, however, the structure requires a tame label:

```
17 if (n.lt.1000) then
    n=2*n
    j=j+1
  goto 17
  endif
```

There are other ways of constructing a Do-While in FORTRAN, but we try to use the above format consistently. You will quickly get used to a statement like 17 if as signaling this structure. Notice that the two final statements are not indented, since they are part of the control structure, not of the inside block.

Do-Until iteration. In Pascal, for example, this is rendered as

```
REPEAT
    n:=n DIV 2;          Pascal's integer divide is DIV.
    k:=k+1;
UNTIL (n=1);
```

In FORTRAN we write

```
19 continue
       n=n/2
       k=k+1
   if (n.ne.1) goto 19
```

Break. In this case, you have a loop that is repeated indefinitely until some condition *tested somewhere in the middle of the loop* (and possibly tested in more than one place) becomes true. At that point you wish to exit the loop and proceed with what comes after it. Standard FORTRAN does not make this structure accessible without labels. We will try to avoid using the structure when we can. Sometimes, however, it is plainly necessary. We do not have the patience to argue with the designers of computer languages over this point. In FORTRAN we write

```
13 continue
       [statements before the test]
       if (···) goto 14
       [statements after the test]
   goto 13
14 continue
```

Here is a program that uses several different iteration structures. One of us was once asked, for a scavenger hunt, to find the date of a Friday the 13th on which the moon was full. This is a program which accomplishes that task, giving incidentally all other Fridays the 13th as a by-product.

```
      PROGRAM badluk
      INTEGER ic,icon,idwk,ifrac,im,iybeg,iyend,iyyy,jd,jday,n,
     *     julday
      REAL TIMZON,frac
      PARAMETER (TIMZON=-5./24.)          Time zone −5 is Eastern Standard Time.
      DATA iybeg,iyend /1900,2000/        The range of dates to be searched.
C     USES flmoon,julday
      write (*,'(1x,a,i5,a,i5)') 'Full moons on Friday the 13th from',
     *     iybeg,' to',iyend
      do 12 iyyy=iybeg,iyend                        Loop over each year,
        do 11 im=1,12                               and each month.
          jday=julday(im,13,iyyy)                   Is the 13th a Friday?
          idwk=mod(jday+1,7)
          if(idwk.eq.5) then
            n=12.37*(iyyy-1900+(im-0.5)/12.)
              This value n is a first approximation to how many full moons have occurred
              since 1900. We will feed it into the phase routine and adjust it up or down until
```

```
                we determine that our desired 13th was or was not a full moon. The variable
                icon signals the direction of adjustment.
              icon=0
1             call flmoon(n,2,jd,frac)        Get date of full moon n.
              ifrac=nint(24.*(frac+TIMZON))   Convert to hours in correct time zone.
              if(ifrac.lt.0)then              Convert from Julian Days beginning at noon
                  jd=jd-1                         to civil days beginning at midnight.
                  ifrac=ifrac+24
              endif
              if(ifrac.gt.12)then
                  jd=jd+1
                  ifrac=ifrac-12
              else
                  ifrac=ifrac+12
              endif
              if(jd.eq.jday)then          Did we hit our target day?
                  write (*,'(/1x,i2,a,i2,a,i4)') im,'/',13,'/',iyyy
                  write (*,'(1x,a,i2,a)') 'Full moon ',ifrac,
*                       ' hrs after midnight (EST).'
                    Don't worry if you are unfamiliar with FORTRAN's esoteric input/output
                    statements; very few programs in this book do any input/output.
                  goto 2                      Part of the break-structure, case of a match.
              else                            Didn't hit it.
                  ic=isign(1,jday-jd)
                  if(ic.eq.-icon) goto 2      Another break, case of no match.
                  icon=ic
                  n=n+ic
              endif
              goto 1
2             continue
            endif
        enddo 11
    enddo 12
    END
```

If you are merely curious, there were (or will be) occurrences of a full moon on Friday the 13th (time zone GMT-5) on: 3/13/1903, 10/13/1905, 6/13/1919, 1/13/1922, 11/13/1970, 2/13/1987, 10/13/2000, 9/13/2019, and 8/13/2049.

Other "standard" structures. Our advice is to avoid them. Every programming language has some number of "goodies" that the designer just couldn't resist throwing in. They seemed like a good idea at the time. Unfortunately they don't stand the *test* of time! Your program becomes difficult to translate into other languages, and difficult to read (because rarely used structures are unfamiliar to the reader). You can almost always accomplish the supposed conveniences of these structures in other ways. Try to do so with the above standard structures, which really *are* standard. If you can't, then use straightforward, unstructured, tests and `goto`'s. This will introduce real (not tame) statement labels, whose very existence will warn the reader to give special thought to the program's control flow.

In `FORTRAN` we consider the ill-advised control structures to be

- assigned `goto` and `assign` statements
- computed `goto` statement
- arithmetic `if` statement

About "Advanced Topics"

Material set in smaller type, like this, signals an "advanced topic," either one outside of the main argument of the chapter, or else one requiring of you more than the usual assumed mathematical background, or else (in a few cases) a discussion that is more speculative or an algorithm that is less well-tested. Nothing important will be lost if you skip the advanced topics on a first reading of the book.

You may have noticed that, by its looping over the months and years, the program `badluk` avoids using any algorithm for converting a Julian Day Number back into a calendar date. A routine for doing just this is not very interesting structurally, but it is occasionally useful:

```
SUBROUTINE caldat(julian,mm,id,iyyy)
INTEGER id,iyyy,julian,mm,IGREG
PARAMETER (IGREG=2299161)
    Inverse of the function julday given above. Here julian is input as a Julian Day Number,
    and the routine outputs mm,id, and iyyy as the month, day, and year on which the specified
    Julian Day started at noon.
INTEGER ja,jalpha,jb,jc,jd,je
if(julian.ge.IGREG)then                          Cross-over to Gregorian Calendar produces
    jalpha=int(((julian-1867216)-0.25)/36524.25)       this correction.
    ja=julian+1+jalpha-int(0.25*jalpha)
else if(julian.lt.0)then                         Make day number positive by adding in-
    ja=julian+36525*(1-julian/36525)                 teger number of Julian centuries, then
else                                                 subtract them off at the end.
    ja=julian
endif
jb=ja+1524
jc=int(6680.+((jb-2439870)-122.1)/365.25)
jd=365*jc+int(0.25*jc)
je=int((jb-jd)/30.6001)
id=jb-jd-int(30.6001*je)
mm=je-1
if(mm.gt.12)mm=mm-12
iyyy=jc-4715
if(mm.gt.2)iyyy=iyyy-1
if(iyyy.le.0)iyyy=iyyy-1
if(julian.lt.0)iyyy=iyyy-100*(1-julian/36525)
return
END
```

(For additional calendrical algorithms, applicable to various historical calendars, see [8].)

Some Habits and Assumed ANSI Extensions

Mentioning a few of our programming habits here will make it easier for you to read the programs in this book.

- We habitually use `m` and `n` to refer to the logical dimensions of a matrix, `mp` and `np` to refer to the physical dimensions. (These important concepts are detailed in §2.0 and Figure 2.0.1.)
- Often, when a subroutine or procedure is to be passed some integer `n`, it needs an internally preset value for the largest possible value that will be passed. We habitually call this `NMAX`, and set it in a `PARAMETER` statement. When we say in a comment, "largest value of `n`," we do not mean to imply that the program will fail algorithmically for larger values, but only that `NMAX` must be altered.
- A number represented by `TINY`, usually a parameter, is supposed to be much smaller than any number of interest to you, but not so small that it underflows. Its use is usually prosaic, to prevent divide checks in some circumstances.

As a matter of typography, the printed FORTRAN programs in this book, if typed into a computer exactly as written, would violate the FORTRAN-77 standard in a few trivial ways. The anomalies, which are *not* present in the machine-readable program distributions, are as follows:

- As already discussed, we use `enddo` followed by the statement label instead of `continue` preceded by the label.
- Standard FORTRAN reads no more than 72 characters on a line and ignores input from column 73 onward. Longer statements are broken up onto "continuation lines." In the printed programs in this book, some lines contain more than 72 characters. When the break to a continuation line is not shown explicitly, it should be inserted when you type the program into a computer.
- In standard FORTRAN, columns 1 through 6 on each line are used variously for (i) statement labels, (ii) signaling a comment line, and (iii) signaling a continuation line. We simplify the format slightly: To the left of the "program left margin," an integer is a statement label (not a "tame label" as described above), an asterisk (`*`) indicates a continuation line, and a "`C`" indicates a comment line. Comment lines shown in this way are generally either `USES` statements (see §1.0), or else "commented-out program lines" that are separately explained in each instance.

A small number of routines in this book require the use of functions that act bitwise on integers, e.g., bitwise "and" or "exclusive or". Unfortunately, although these functions are available in virtually all modern FORTRAN implementations, they are not a part of the FORTRAN-77 standard. Even more unfortunate is the fact that there are two different naming conventions in widespread use. We use the names `iand(i,j)`, `ior(i,j)`, `not(i)`, `ieor(i,j)`, and `ishft(i,j)`, for *and*, *or*, *not*, *exclusive-or*, and *left-shift*, respectively, as well as the subroutines `ibset(i,j)`, `ibclr(i,j)`, and the logical function `btest(i,j)` for *bit-set*, *bit-clear*, and *bit-test*. Some (mainly UNIX) FORTRAN compilers use a different set of names, with the following correspondences:

Us...	Them...	
`iand(i,j)`	= `and(i,j)`	
`ior(i,j)`	= `or(i,j)`	
`not(i)`	= `not(i)`	
`ieor(i,j)`	= `xor(i,j)`	
`ishft(i,j)`	= `lshft(i,j)`	
`ibset(i,j)`	= `bis(j,i)`	Note reversed arguments!
`ibclr(i,j)`	= `bic(j,i)`	Ditto!
`btest(i,j)`	= `bit(j,i)`	Ditto!

If you are one of "Them," you can either modify the small number of programs affected (e.g., by inserting FORTRAN statement function definitions at the beginning of the routines), or else link to an object file into which you have compiled the trivial functions that define "our" names in terms of "yours," as in the above table. Standards really are important!

Hexadecimal constants, for which there is no standard notation in FORTRAN compilers, occur at three places in Chapter 7: a program fragment at the end of §7.1,

and routines psdes and ran4 in §7.5. We use a notation like Z'3F800000', which is consistent with the new FORTRAN-90 standard, but you may need to change this to, e.g., x'3f800000', '3F800000'X, or even 16#3F800000. In extremis, you can convert the hex values to decimal integers; but note that most compilers will require a *negative* decimal integer as the value of a hex constant with its high-order bit set.

As already mentioned in §1.0, the notation a(1:m), in program comments and in the text, denotes the array element range a(1), a(2), ..., a(m). Likewise, notations like b(2:7) or c(1:m,1:n) are to be interpreted as denoting ranges of array indices.

CITED REFERENCES AND FURTHER READING:

Kernighan, B.W. 1978, *The Elements of Programming Style* (New York: McGraw-Hill). [1]

Yourdon, E. 1975, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Meissner, L.P. and Organick, E.I. 1980, *Fortran 77 Featuring Structured Programming* (Reading, MA: Addison-Wesley). [3]

Hoare, C.A.R. 1981, *Communications of the ACM*, vol. 24, pp. 75–83.

Wirth, N. 1983, *Programming in Modula-2*, 3rd ed. (New York: Springer-Verlag). [4]

Stroustrup, B. 1986, *The C++ Programming Language* (Reading, MA: Addison-Wesley). [5]

Borland International, Inc. 1989, *Turbo Pascal 5.5 Object-Oriented Programming Guide* (Scotts Valley, CA: Borland International). [6]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [7]

Hatcher, D.A. 1984, *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53–55; see also *op. cit.* 1985, vol. 26, pp. 151–155, and 1986, vol. 27, pp. 506–507. [8]

1.2 Error, Accuracy, and Stability

Although we assume no prior training of the reader in formal numerical analysis, we will need to presume a common understanding of a few key concepts. We will define these briefly in this section.

Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of *bits* (binary digits) or *bytes* (groups of 8 bits). Almost all computers allow the programmer a choice among several different such *representations* or *data types*. Data types can differ in the number of bits utilized (the *wordlength*), but also in the more fundamental respect of whether the stored number is represented in *fixed-point* (also called *integer*) or *floating-point* (also called *real*) format.

A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that (i) the answer is not outside the range of (usually, signed) integers that can be represented, and (ii) that division is interpreted as producing an integer result, throwing away any integer remainder.

and routines psdes and ran4 in §7.5. We use a notation like Z'3F800000', which is consistent with the new FORTRAN-90 standard, but you may need to change this to, e.g., x'3f800000', '3F800000'X, or even 16#3F800000. In extremis, you can convert the hex values to decimal integers; but note that most compilers will require a *negative* decimal integer as the value of a hex constant with its high-order bit set.

As already mentioned in §1.0, the notation a(1:m), in program comments and in the text, denotes the array element range a(1), a(2), ..., a(m). Likewise, notations like b(2:7) or c(1:m,1:n) are to be interpreted as denoting ranges of array indices.

CITED REFERENCES AND FURTHER READING:

Kernighan, B.W. 1978, *The Elements of Programming Style* (New York: McGraw-Hill). [1]

Yourdon, E. 1975, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Meissner, L.P. and Organick, E.I. 1980, *Fortran 77 Featuring Structured Programming* (Reading, MA: Addison-Wesley). [3]

Hoare, C.A.R. 1981, *Communications of the ACM*, vol. 24, pp. 75–83.

Wirth, N. 1983, *Programming in Modula-2*, 3rd ed. (New York: Springer-Verlag). [4]

Stroustrup, B. 1986, *The C++ Programming Language* (Reading, MA: Addison-Wesley). [5]

Borland International, Inc. 1989, *Turbo Pascal 5.5 Object-Oriented Programming Guide* (Scotts Valley, CA: Borland International). [6]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [7]

Hatcher, D.A. 1984, *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53–55; see also *op. cit.* 1985, vol. 26, pp. 151–155, and 1986, vol. 27, pp. 506–507. [8]

1.2 Error, Accuracy, and Stability

Although we assume no prior training of the reader in formal numerical analysis, we will need to presume a common understanding of a few key concepts. We will define these briefly in this section.

Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of *bits* (binary digits) or *bytes* (groups of 8 bits). Almost all computers allow the programmer a choice among several different such *representations* or *data types*. Data types can differ in the number of bits utilized (the *wordlength*), but also in the more fundamental respect of whether the stored number is represented in *fixed-point* (also called *integer*) or *floating-point* (also called *real*) format.

A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that (i) the answer is not outside the range of (usually, signed) integers that can be represented, and (ii) that division is interpreted as producing an integer result, throwing away any integer remainder.

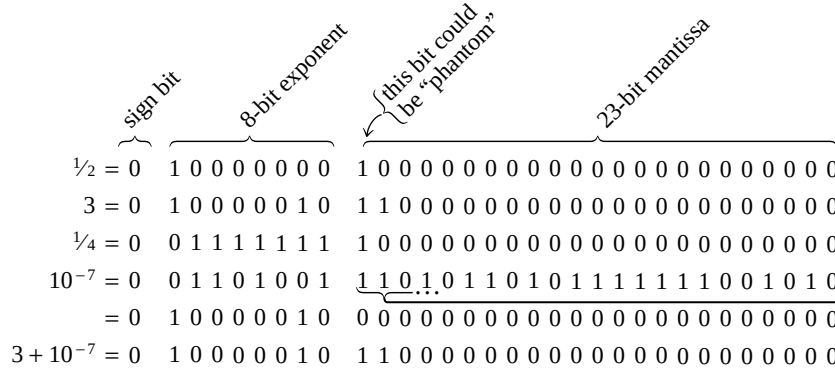


Figure 1.2.1. Floating point representations of numbers in a typical 32-bit (4-byte) format. (a) The number $1/2$ (note the bias in the exponent); (b) the number 3; (c) the number $1/4$; (d) the number 10^{-7}, represented to machine accuracy; (e) the same number 10^{-7}, but shifted so as to have the same exponent as the number 3; with this shifting, all significance is lost and 10^{-7} becomes zero; shifting to a common exponent must occur before two numbers can be added; (f) sum of the numbers $3 + 10^{-7}$, which equals 3 to machine accuracy. Even though 10^{-7} can be represented accurately by itself, it cannot accurately be added to a much larger number.

In floating-point representation, a number is represented internally by a sign bit s (interpreted as plus or minus), an exact integer exponent e, and an exact positive integer mantissa M. Taken together these represent the number

$$s \times M \times B^{e-E} \tag{1.2.1}$$

where B is the base of the representation (usually $B = 2$, but sometimes $B = 16$), and E is the *bias* of the exponent, a fixed integer constant for any given machine and representation. An example is shown in Figure 1.2.1.

Several floating-point bit patterns can represent the same number. If $B = 2$, for example, a mantissa with leading (high-order) zero bits can be left-shifted, i.e., multiplied by a power of 2, if the exponent is decreased by a compensating amount. Bit patterns that are "as left-shifted as they can be" are termed *normalized*. Most computers always produce normalized results, since these don't waste any bits of the mantissa and thus allow a greater accuracy of the representation. Since the high-order bit of a properly normalized mantissa (when $B = 2$) is *always* one, some computers don't store this bit at all, giving one extra bit of significance.

Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented (i.e., have exact values in the form of equation 1.2.1). For example, two floating numbers are added by first right-shifting (dividing by two) the mantissa of the smaller (in magnitude) one, simultaneously increasing its exponent, until the two operands have the same exponent. Low-order (least significant) bits of the smaller operand are lost by this shifting. If the two operands differ too greatly in magnitude, then the smaller operand is effectively replaced by zero, since it is right-shifted to oblivion.

The smallest (in magnitude) floating-point number which, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the *machine accuracy* ϵ_m. A typical computer with $B = 2$ and a 32-bit wordlength has ϵ_m around 3×10^{-8}. (A more detailed discussion of machine

characteristics, and a program to determine them, is given in §20.1.) Roughly speaking, the machine accuracy ϵ_m is the fractional accuracy to which floating-point numbers are represented, corresponding to a change of one in the least significant bit of the mantissa. Pretty much any arithmetic operation among floating numbers should be thought of as introducing an additional fractional error of at least ϵ_m. This type of error is called *roundoff error*.

It is important to understand that ϵ_m is not the smallest floating-point number that can be represented on a machine. *That* number depends on how many bits there are in the exponent, while ϵ_m depends on how many bits there are in the mantissa.

Roundoff errors accumulate with increasing amounts of calculation. If, in the course of obtaining a calculated value, you perform N such arithmetic operations, you *might* be so lucky as to have a total roundoff error on the order of $\sqrt{N}\epsilon_m$, if the roundoff errors come in randomly up or down. (The square root comes from a random-walk.) However, this estimate can be very badly off the mark for two reasons:

(i) It very frequently happens that the regularities of your calculation, or the peculiarities of your computer, cause the roundoff errors to accumulate preferentially in one direction. In this case the total will be of order $N\epsilon_m$.

(ii) Some especially unfavorable occurrences can vastly increase the roundoff error of single operations. Generally these can be traced to the subtraction of two very nearly equal numbers, giving a result whose only significant bits are those (few) low-order ones in which the operands differed. You might think that such a "coincidental" subtraction is unlikely to occur. Not always so. Some mathematical expressions magnify its probability of occurrence tremendously. For example, in the familiar formula for the solution of a quadratic equation,

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{1.2.2}$$

the addition becomes delicate and roundoff-prone whenever $ac \ll b^2$. (In §5.6 we will learn how to avoid the problem in this particular case.)

Roundoff error is a characteristic of computer hardware. There is another, different, kind of error that is a characteristic of the program or algorithm used, independent of the hardware on which the program is executed. Many numerical algorithms compute "discrete" approximations to some desired "continuous" quantity. For example, an integral is evaluated numerically by computing a function at a discrete set of points, rather than at "every" point. Or, a function may be evaluated by summing a finite number of leading terms in its infinite series, rather than all infinity terms. In cases like this, there is an adjustable parameter, e.g., the number of points or of terms, such that the "true" answer is obtained only when that parameter goes to infinity. Any practical calculation is done with a finite, but sufficiently large, choice of that parameter.

The discrepancy between the true answer and the answer obtained in a practical calculation is called the *truncation error*. Truncation error would persist even on a hypothetical, "perfect" computer that had an infinitely accurate representation and no roundoff error. As a general rule there is not much that a programmer can do about roundoff error, other than to choose algorithms that do not magnify it unnecessarily (see discussion of "stability" below). Truncation error, on the other hand, is entirely under the programmer's control. In fact, it is only a slight exaggeration to say

that clever minimization of truncation error is practically the entire content of the field of numerical analysis!

Most of the time, truncation error and roundoff error do not strongly interact with one another. A calculation can be imagined as having, first, the truncation error that it would have if run on an infinite-precision computer, "plus" the roundoff error associated with the number of operations performed.

Sometimes, however, an otherwise attractive method can be *unstable*. This means that any roundoff error that becomes "mixed into" the calculation at an early stage is successively magnified until it comes to swamp the true answer. An unstable method would be useful on a hypothetical, perfect computer; but in this imperfect world it is necessary for us to require that algorithms be stable — or if unstable that we use them with great caution.

Here is a simple, if somewhat artificial, example of an unstable algorithm: Suppose that it is desired to calculate all integer powers of the so-called "Golden Mean," the number given by

$$\phi \equiv \frac{\sqrt{5}-1}{2} \approx 0.61803398 \tag{1.2.3}$$

It turns out (you can easily verify) that the powers ϕ^n satisfy a simple recursion relation,

$$\phi^{n+1} = \phi^{n-1} - \phi^n \tag{1.2.4}$$

Thus, knowing the first two values $\phi^0 = 1$ and $\phi^1 = 0.61803398$, we can successively apply (1.2.4) performing only a single subtraction, rather than a slower multiplication by ϕ, at each stage.

Unfortunately, the recurrence (1.2.4) also has *another* solution, namely the value $-\frac{1}{2}(\sqrt{5}+1)$. Since the recurrence is linear, and since this undesired solution has magnitude greater than unity, any small admixture of it introduced by roundoff errors will grow exponentially. On a typical machine with 32-bit wordlength, (1.2.4) starts to give completely wrong answers by about $n = 16$, at which point ϕ^n is down to only 10^{-4}. The recurrence (1.2.4) is *unstable*, and cannot be used for the purpose stated.

We will encounter the question of stability in many more sophisticated guises, later in this book.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 1.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 2.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §1.3.

Wilkinson, J.H. 1964, *Rounding Errors in Algebraic Processes* (Englewood Cliffs, NJ: Prentice-Hall).

Chapter 2. Solution of Linear Algebraic Equations

2.0 Introduction

A set of linear algebraic equations looks like this:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N &= b_3 \\ \cdots \qquad\qquad & \cdots \\ a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N &= b_M \end{aligned} \tag{2.0.1}$$

Here the N unknowns x_j, $j = 1, 2, \ldots, N$ are related by M equations. The coefficients a_{ij} with $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$ are known numbers, as are the *right-hand side* quantities b_i, $i = 1, 2, \ldots, M$.

Nonsingular versus Singular Sets of Equations

If $N = M$ then there are as many equations as unknowns, and there is a good chance of solving for a unique solution set of x_j's. Analytically, there can fail to be a unique solution if one or more of the M equations is a linear combination of the others, a condition called *row degeneracy*, or if all equations contain certain variables only in exactly the same linear combination, called *column degeneracy*. (For square matrices, a row degeneracy implies a column degeneracy, and vice versa.) A set of equations that is degenerate is called *singular*. We will consider singular matrices in some detail in §2.6.

Numerically, at least two additional things can go wrong:

- While not exact linear combinations of each other, some of the equations may be so close to linearly dependent that roundoff errors in the machine render them linearly dependent at some stage in the solution process. In this case your numerical procedure will fail, and it can tell you that it has failed.

- Accumulated roundoff errors in the solution process can swamp the true solution. This problem particularly emerges if N is too large. The numerical procedure does not fail algorithmically. However, it returns a set of x's that are wrong, as can be discovered by direct substitution back into the original equations. The closer a set of equations is to being singular, the more likely this is to happen, since increasingly close cancellations will occur during the solution. In fact, the preceding item can be viewed as the special case where the loss of significance is unfortunately total.

Much of the sophistication of complicated "linear equation-solving packages" is devoted to the detection and/or correction of these two pathologies. As you work with large linear sets of equations, you will develop a feeling for when such sophistication is needed. It is difficult to give any firm guidelines, since there is no such thing as a "typical" linear problem. But here is a rough idea: Linear sets with N as large as 20 or 50 can be routinely solved in single precision (32 bit floating representations) without resorting to sophisticated methods, *if* the equations are not close to singular. With double precision (60 or 64 bits), this number can readily be extended to N as large as several hundred, after which point the limiting factor is generally machine time, not accuracy.

Even larger linear sets, N in the thousands or greater, can be solved when the coefficients are sparse (that is, mostly zero), by methods that take advantage of the sparseness. We discuss this further in §2.7.

At the other end of the spectrum, one seems just as often to encounter linear problems which, by their underlying nature, are close to singular. In this case, you *might* need to resort to sophisticated methods even for the case of $N = 10$ (though rarely for $N = 5$). Singular value decomposition (§2.6) is a technique that can sometimes turn singular problems into nonsingular ones, in which case additional sophistication becomes unnecessary.

Matrices

Equation (2.0.1) can be written in matrix form as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.0.2}$$

Here the raised dot denotes matrix multiplication, $\mathbf{A}$ is the matrix of coefficients, and $\mathbf{b}$ is the right-hand side written as a column vector,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ & \cdots & & \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_M \end{bmatrix} \tag{2.0.3}$$

By convention, the first index on an element a_{ij} denotes its row, the second index its column. A computer will store the matrix $\mathbf{A}$ as a two-dimensional array. However, computer memory is numbered sequentially by its address, and so is intrinsically one-dimensional. Therefore the two-dimensional array $\mathbf{A}$ will, at the hardware level, either be *stored by columns* in the order

$$a_{11}, a_{21}, \dots, a_{M1},\ a_{12}, a_{22}, \dots, a_{M2},\ \dots,\ a_{1N}, a_{2N}, \dots a_{MN}$$

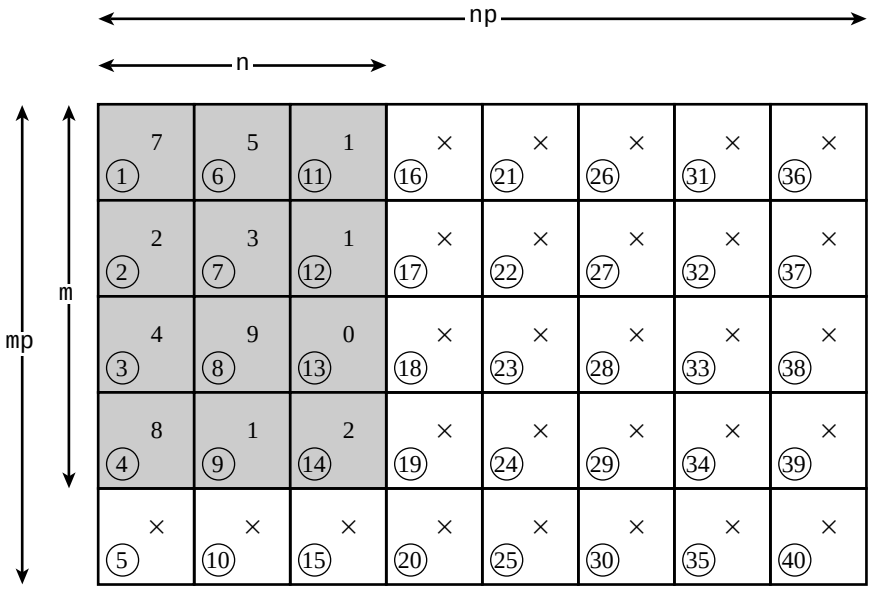


Figure 2.0.1. A matrix of logical dimension `m` by `n` is stored in an array of physical dimension `mp` by `np`. Locations marked by "`x`" contain extraneous information which may be left over from some previous use of the physical array. Circled numbers show the actual ordering of the array in computer memory, not usually relevant to the programmer. Note, however, that the logical array does not occupy consecutive memory locations. To locate an `(i,j)` element correctly, a subroutine must be told `mp` and `np`, not just `i` and `j`.

or else *stored by rows* in the order

$$a_{11}, a_{12}, \ldots, a_{1N},\ a_{21}, a_{22}, \ldots, a_{2N},\ \ldots,\ a_{M1}, a_{M2}, \ldots a_{MN}$$

`FORTRAN` always stores by columns, and user programs are generally allowed to exploit this fact to their advantage. By contrast, `C`, `Pascal`, and other languages generally store by rows. Note one confusing point in the terminology, that a matrix which is stored by columns (as in `FORTRAN`) has its *row* (i.e., first) index changing most rapidly as one goes linearly through memory, the opposite of a car's odometer!

For most purposes you don't *need* to know what the order of storage is, since you reference an element by its two-dimensional address: a_{34} = `a(3,4)`. It is, however, *essential* that you understand the difference between an array's *physical dimensions* and its *logical dimensions*. When you pass an array to a subroutine, you must, in general, tell the subroutine *both* of these dimensions. The distinction between them is this: It may happen that you have a 4×4 matrix stored in an array dimensioned as 10×10. This occurs most frequently in practice when you have dimensioned to the largest expected value of N, but are at the moment considering a value of N smaller than that largest possible one. In the example posed, the 16 elements of the matrix do not occupy 16 consecutive memory locations. Rather they are spread out among the 100 dimensioned locations of the array as if the whole 10×10 matrix were filled. Figure 2.0.1 shows an additional example.

If you have a subroutine to invert a matrix, its call might typically look like this:

```
call matinv(a,ai,n,np)
```

Here the subroutine has to be told both the logical size of the matrix that you want to invert (here $\texttt{n} = 4$), and the physical size of the array in which it is stored (here $\texttt{np} = 10$).

This seems like a trivial point, and we are sorry to belabor it. But it turns out that *most* reported failures of standard linear equation and matrix manipulation packages are due to user errors in passing inappropriate logical or physical dimensions!

Tasks of Computational Linear Algebra

We will consider the following tasks as falling in the general purview of this chapter:

- Solution of the matrix equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for an unknown vector $\mathbf{x}$, where $\mathbf{A}$ is a square matrix of coefficients, raised dot denotes matrix multiplication, and $\mathbf{b}$ is a known right-hand side vector (§2.1–§2.10).
- Solution of more than one matrix equation $\mathbf{A} \cdot \mathbf{x}_j = \mathbf{b}_j$, for a set of vectors $\mathbf{x}_j$, $j = 1, 2, \ldots$, each corresponding to a different, known right-hand side vector $\mathbf{b}_j$. In this task the key simplification is that the matrix $\mathbf{A}$ is held constant, while the right-hand sides, the $\mathbf{b}$'s, are changed (§2.1–§2.10).
- Calculation of the matrix $\mathbf{A}^{-1}$ which is the matrix inverse of a square matrix $\mathbf{A}$, i.e., $\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{1}$, where $\mathbf{1}$ is the identity matrix (all zeros except for ones on the diagonal). This task is equivalent, for an $N \times N$ matrix $\mathbf{A}$, to the previous task with N different $\mathbf{b}_j$'s ($j = 1, 2, \ldots, N$), namely the unit vectors ($\mathbf{b}_j$ = all zero elements except for 1 in the jth component). The corresponding $\mathbf{x}$'s are then the columns of the matrix inverse of $\mathbf{A}$ (§2.1 and §2.3).
- Calculation of the determinant of a square matrix $\mathbf{A}$ (§2.3).

If $M < N$, or if $M = N$ but the equations are degenerate, then there are effectively fewer equations than unknowns. In this case there can be either no solution, or else more than one solution vector $\mathbf{x}$. In the latter event, the solution space consists of a particular solution $\mathbf{x}_p$ added to any linear combination of (typically) $N - M$ vectors (which are said to be in the nullspace of the matrix $\mathbf{A}$). The task of finding the solution space of $\mathbf{A}$ involves

- Singular value decomposition of a matrix $\mathbf{A}$.

This subject is treated in §2.6.

In the opposite case there are more equations than unknowns, $M > N$. When this occurs there is, in general, no solution vector $\mathbf{x}$ to equation (2.0.1), and the set of equations is said to be *overdetermined*. It happens frequently, however, that the best "compromise" solution is sought, the one that comes closest to satisfying all equations simultaneously. If closeness is defined in the least-squares sense, i.e., that the sum of the squares of the differences between the left- and right-hand sides of equation (2.0.1) be minimized, then the overdetermined linear problem reduces to a

(usually) solvable linear problem, called the

- Linear least-squares problem.

The reduced set of equations to be solved can be written as the $N \times N$ set of equations

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = (\mathbf{A}^T \cdot \mathbf{b}) \tag{2.0.4}$$

where $\mathbf{A}^T$ denotes the transpose of the matrix $\mathbf{A}$. Equations (2.0.4) are called the *normal equations* of the linear least-squares problem. There is a close connection between singular value decomposition and the linear least-squares problem, and the latter is also discussed in §2.6. You should be warned that direct solution of the normal equations (2.0.4) is not generally the best way to find least-squares solutions.

Some other topics in this chapter include

- Iterative improvement of a solution (§2.5)
- Various special forms: symmetric positive-definite (§2.9), tridiagonal (§2.4), band diagonal (§2.4), Toeplitz (§2.8), Vandermonde (§2.8), sparse (§2.7)
- Strassen's "fast matrix inversion" (§2.11).

Standard Subroutine Packages

We cannot hope, in this chapter or in this book, to tell you everything there is to know about the tasks that have been defined above. In many cases you will have no alternative but to use sophisticated black-box program packages. Several good ones are available. LINPACK was developed at Argonne National Laboratories and deserves particular mention because it is published, documented, and available for free use. A successor to LINPACK, LAPACK, is now becoming available. Packages available commercially include those in the IMSL and NAG libraries.

You should keep in mind that the sophisticated packages are designed with very large linear systems in mind. They therefore go to great effort to minimize not only the number of operations, but also the required storage. Routines for the various tasks are usually provided in several versions, corresponding to several possible simplifications in the form of the input coefficient matrix: symmetric, triangular, banded, positive definite, etc. If you have a large matrix in one of these forms, you should certainly take advantage of the increased efficiency provided by these different routines, and not just use the form provided for general matrices.

There is also a great watershed dividing routines that are *direct* (i.e., execute in a predictable number of operations) from routines that are *iterative* (i.e., attempt to converge to the desired answer in however many steps are necessary). Iterative methods become preferable when the battle against loss of significance is in danger of being lost, either due to large N or because the problem is close to singular. We will treat iterative methods only incompletely in this book, in §2.7 and in Chapters 18 and 19. These methods are important, but mostly beyond our scope. We will, however, discuss in detail a technique which is on the borderline between direct and iterative methods, namely the iterative improvement of a solution that has been obtained by direct methods (§2.5).

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press).

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.).

Coleman, T.F., and Van Loan, C. 1988, *Handbook for Matrix Computations* (Philadelphia: S.I.A.M.).

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag).

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 9.

2.1 Gauss-Jordan Elimination

For inverting a matrix, *Gauss-Jordan elimination* is about as efficient as any other method. For solving sets of linear equations, Gauss-Jordan elimination produces *both* the solution of the equations for one or more right-hand side vectors $\mathbf{b}$, and also the matrix inverse $\mathbf{A}^{-1}$. However, its principal weaknesses are (i) that it requires all the right-hand sides to be stored and manipulated at the same time, and (ii) that when the inverse matrix is *not* desired, Gauss-Jordan is three times slower than the best alternative technique for solving a single linear set (§2.3). The method's principal strength is that it is as stable as any other direct method, perhaps even a bit more stable when full pivoting is used (see below).

If you come along later with an additional right-hand side vector, you can multiply it by the inverse matrix, of course. This does give an answer, but one that is quite susceptible to roundoff error, not nearly as good as if the new vector had been included with the set of right-hand side vectors in the first instance.

For these reasons, Gauss-Jordan elimination should usually not be your method of first choice, either for solving linear equations or for matrix inversion. The decomposition methods in §2.3 are better. Why do we give you Gauss-Jordan at all? Because it is straightforward, understandable, solid as a rock, and an exceptionally good "psychological" backup for those times that something is going wrong and you think it *might* be your linear-equation solver.

Some people believe that the backup is more than psychological, that Gauss-Jordan elimination is an "independent" numerical method. This turns out to be mostly myth. Except for the relatively minor differences in pivoting, described below, the actual sequence of operations performed in Gauss-Jordan elimination is very closely related to that performed by the routines in the next two sections.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press).

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.).

Coleman, T.F., and Van Loan, C. 1988, *Handbook for Matrix Computations* (Philadelphia: S.I.A.M.).

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag).

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 9.

2.1 Gauss-Jordan Elimination

For inverting a matrix, *Gauss-Jordan elimination* is about as efficient as any other method. For solving sets of linear equations, Gauss-Jordan elimination produces *both* the solution of the equations for one or more right-hand side vectors $\mathbf{b}$, and also the matrix inverse $\mathbf{A}^{-1}$. However, its principal weaknesses are (i) that it requires all the right-hand sides to be stored and manipulated at the same time, and (ii) that when the inverse matrix is *not* desired, Gauss-Jordan is three times slower than the best alternative technique for solving a single linear set (§2.3). The method's principal strength is that it is as stable as any other direct method, perhaps even a bit more stable when full pivoting is used (see below).

If you come along later with an additional right-hand side vector, you can multiply it by the inverse matrix, of course. This does give an answer, but one that is quite susceptible to roundoff error, not nearly as good as if the new vector had been included with the set of right-hand side vectors in the first instance.

For these reasons, Gauss-Jordan elimination should usually not be your method of first choice, either for solving linear equations or for matrix inversion. The decomposition methods in §2.3 are better. Why do we give you Gauss-Jordan at all? Because it is straightforward, understandable, solid as a rock, and an exceptionally good "psychological" backup for those times that something is going wrong and you think it *might* be your linear-equation solver.

Some people believe that the backup is more than psychological, that Gauss-Jordan elimination is an "independent" numerical method. This turns out to be mostly myth. Except for the relatively minor differences in pivoting, described below, the actual sequence of operations performed in Gauss-Jordan elimination is very closely related to that performed by the routines in the next two sections.

For clarity, and to avoid writing endless ellipses $(\cdots)$ we will write out equations only for the case of four equations and four unknowns, and with three different right-hand side vectors that are known in advance. You can write bigger matrices and extend the equations to the case of $N \times N$ matrices, with M sets of right-hand side vectors, in completely analogous fashion. The routine implemented below is, of course, general.

Elimination on Column-Augmented Matrices

Consider the linear matrix equation

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \left[\begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{pmatrix} \sqcup \begin{pmatrix} x_{12} \\ x_{22} \\ x_{32} \\ x_{42} \end{pmatrix} \sqcup \begin{pmatrix} x_{13} \\ x_{23} \\ x_{33} \\ x_{43} \end{pmatrix} \sqcup \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{pmatrix} \right]$$

$$= \left[\begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{pmatrix} \sqcup \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ b_{42} \end{pmatrix} \sqcup \begin{pmatrix} b_{13} \\ b_{23} \\ b_{33} \\ b_{43} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \tag{2.1.1}$$

Here the raised dot $(\cdot)$ signifies matrix multiplication, while the operator $\sqcup$ just signifies column augmentation, that is, removing the abutting parentheses and making a wider matrix out of the operands of the $\sqcup$ operator.

It should not take you long to write out equation (2.1.1) and to see that it simply states that x_{ij} is the ith component ($i = 1, 2, 3, 4$) of the vector solution of the jth right-hand side ($j = 1, 2, 3$), the one whose coefficients are $b_{ij}, i = 1, 2, 3, 4$; and that the matrix of unknown coefficients y_{ij} is the inverse matrix of a_{ij}. In other words, the matrix solution of

$$[\mathbf{A}] \cdot [\mathbf{x}_1 \sqcup \mathbf{x}_2 \sqcup \mathbf{x}_3 \sqcup \mathbf{Y}] = [\mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{b}_3 \sqcup \mathbf{1}] \tag{2.1.2}$$

where $\mathbf{A}$ and $\mathbf{Y}$ are square matrices, the $\mathbf{b}_i$'s and $\mathbf{x}_i$'s are column vectors, and $\mathbf{1}$ is the identity matrix, simultaneously solves the linear sets

$$\mathbf{A} \cdot \mathbf{x}_1 = \mathbf{b}_1 \qquad \mathbf{A} \cdot \mathbf{x}_2 = \mathbf{b}_2 \qquad \mathbf{A} \cdot \mathbf{x}_3 = \mathbf{b}_3 \tag{2.1.3}$$

and

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{1} \tag{2.1.4}$$

Now it is also elementary to verify the following facts about (2.1.1):

- Interchanging any two *rows* of $\mathbf{A}$ and the corresponding *rows* of the $\mathbf{b}$'s and of $\mathbf{1}$, does not change (or scramble in any way) the solution $\mathbf{x}$'s and $\mathbf{Y}$. Rather, it just corresponds to writing the same set of linear equations in a different order.
- Likewise, the solution set is unchanged and in no way scrambled if we replace any row in $\mathbf{A}$ by a linear combination of itself and any other row, as long as we do the same linear combination of the rows of the $\mathbf{b}$'s and $\mathbf{1}$ (which then is no longer the identity matrix, of course).

- Interchanging any two *columns* of **A** gives the same solution set only if we simultaneously interchange corresponding *rows* of the **x**'s and of **Y**. In other words, this interchange scrambles the order of the rows in the solution. If we do this, we will need to unscramble the solution by restoring the rows to their original order.

Gauss-Jordan elimination uses one or more of the above operations to reduce the matrix **A** to the identity matrix. When this is accomplished, the right-hand side becomes the solution set, as one sees instantly from (2.1.2).

Pivoting

In "Gauss-Jordan elimination with no pivoting," only the second operation in the above list is used. The first row is divided by the element a_{11} (this being a trivial linear combination of the first row with any other row — zero coefficient for the other row). Then the right amount of the first row is subtracted from each other row to make all the remaining a_{i1}'s zero. The first column of **A** now agrees with the identity matrix. We move to the second column and divide the second row by a_{22}, then subtract the right amount of the second row from rows 1, 3, and 4, so as to make their entries in the second column zero. The second column is now reduced to the identity form. And so on for the third and fourth columns. As we do these operations to **A**, we of course also do the corresponding operations to the **b**'s and to **1** (which by now no longer resembles the identity matrix in any way!).

Obviously we will run into trouble if we ever encounter a zero element on the (then current) diagonal when we are going to divide by the diagonal element. (The element that we divide by, incidentally, is called the *pivot element* or *pivot*.) Not so obvious, but true, is the fact that Gauss-Jordan elimination with no pivoting (no use of the first or third procedures in the above list) is numerically unstable in the presence of any roundoff error, even when a zero pivot is not encountered. You must *never* do Gauss-Jordan elimination (or Gaussian elimination, see below) without pivoting!

So what *is* this magic pivoting? Nothing more than interchanging rows (*partial pivoting*) or rows and columns (*full pivoting*), so as to put a particularly desirable element in the diagonal position from which the pivot is about to be selected. Since we don't want to mess up the part of the identity matrix that we have already built up, we can choose among elements that are both (i) on rows below (or on) the one that is about to be normalized, and also (ii) on columns to the right (or on) the column we are about to eliminate. Partial pivoting is easier than full pivoting, because we don't have to keep track of the permutation of the solution vector. Partial pivoting makes available as pivots only the elements already in the correct column. It turns out that partial pivoting is "almost" as good as full pivoting, in a sense that can be made mathematically precise, but which need not concern us here (for discussion and references, see [1]). To show you both variants, we do full pivoting in the routine in this section, partial pivoting in §2.3.

We have to state how to recognize a particularly desirable pivot when we see one. The answer to this is not completely known theoretically. It is known, both theoretically and in practice, that simply picking the largest (in magnitude) available element as the pivot is a very good choice. A curiosity of this procedure, however, is that the choice of pivot will depend on the original scaling of the equations. If we take the third linear equation in our original set and multiply it by a factor of a million, it

is almost guaranteed that it will contribute the first pivot; yet the underlying solution of the equations is not changed by this multiplication! One therefore sometimes sees routines which choose as pivot that element which *would* have been largest if the original equations had all been scaled to have their largest coefficient normalized to unity. This is called *implicit pivoting*. There is some extra bookkeeping to keep track of the scale factors by which the rows would have been multiplied. (The routines in §2.3 include implicit pivoting, but the routine in this section does not.)

Finally, let us consider the storage requirements of the method. With a little reflection you will see that at every stage of the algorithm, *either* an element of **A** is predictably a one or zero (if it is already in a part of the matrix that has been reduced to identity form) *or else* the exactly corresponding element of the matrix that started as **1** is predictably a one or zero (if its mate in **A** has not been reduced to the identity form). Therefore the matrix **1** does not have to exist as separate storage: The matrix inverse of **A** is gradually built up in **A** as the original **A** is destroyed. Likewise, the solution vectors **x** can gradually replace the right-hand side vectors **b** and share the same storage, since after each column in **A** is reduced, the corresponding row entry in the **b**'s is never again used.

Here is the routine for Gauss-Jordan elimination with full pivoting:

```
      SUBROUTINE gaussj(a,n,np,b,m,mp)
      INTEGER m,mp,n,np,NMAX
      REAL a(np,np),b(np,mp)
      PARAMETER (NMAX=50)
          Linear equation solution by Gauss-Jordan elimination, equation (2.1.1) above. a(1:n,1:n)
          is an input matrix stored in an array of physical dimensions np by np. b(1:n,1:m) is an in-
          put matrix containing the m right-hand side vectors, stored in an array of physical dimensions
          np by mp. On output, a(1:n,1:n) is replaced by its matrix inverse, and b(1:n,1:m) is
          replaced by the corresponding set of solution vectors.
          Parameter: NMAX is the largest anticipated value of n.
      INTEGER i,icol,irow,j,k,l,ll,indxc(NMAX),indxr(NMAX),
     *      ipiv(NMAX)                      The integer arrays ipiv, indxr, and indxc are used
      REAL big,dum,pivinv                       for bookkeeping on the pivoting.
      do 11 j=1,n
          ipiv(j)=0
      enddo 11
      do 22 i=1,n                             This is the main loop over the columns to be re-
          big=0.                                  duced.
          do 13 j=1,n                         This is the outer loop of the search for a pivot ele-
              if(ipiv(j).ne.1)then                ment.
                  do 12 k=1,n
                      if (ipiv(k).eq.0) then
                          if (abs(a(j,k)).ge.big)then
                              big=abs(a(j,k))
                              irow=j
                              icol=k
                          endif
                      else if (ipiv(k).gt.1) then
                          pause 'singular matrix in gaussj'
                      endif
                  enddo 12
              endif
          enddo 13
          ipiv(icol)=ipiv(icol)+1
            We now have the pivot element, so we interchange rows, if needed, to put the pivot
            element on the diagonal. The columns are not physically interchanged, only relabeled:
```

```
      indxc(i), the column of the ith pivot element, is the ith column that is reduced, while
      indxr(i) is the row in which that pivot element was originally located. If indxr(i) ≠
      indxc(i) there is an implied column interchange. With this form of bookkeeping, the
      solution b's will end up in the correct order, and the inverse matrix will be scrambled
      by columns.
    if (irow.ne.icol) then
        do 14 l=1,n
            dum=a(irow,l)
            a(irow,l)=a(icol,l)
            a(icol,l)=dum
        enddo 14
        do 15 l=1,m
            dum=b(irow,l)
            b(irow,l)=b(icol,l)
            b(icol,l)=dum
        enddo 15
    endif
    indxr(i)=irow                    We are now ready to divide the pivot row by the pivot
    indxc(i)=icol                        element, located at irow and icol.
    if (a(icol,icol).eq.0.) pause 'singular matrix in gaussj'
    pivinv=1./a(icol,icol)
    a(icol,icol)=1.
    do 16 l=1,n
        a(icol,l)=a(icol,l)*pivinv
    enddo 16
    do 17 l=1,m
        b(icol,l)=b(icol,l)*pivinv
    enddo 17
    do 21 ll=1,n                     Next, we reduce the rows...
        if(ll.ne.icol)then           ...except for the pivot one, of course.
            dum=a(ll,icol)
            a(ll,icol)=0.
            do 18 l=1,n
                a(ll,l)=a(ll,l)-a(icol,l)*dum
            enddo 18
            do 19 l=1,m
                b(ll,l)=b(ll,l)-b(icol,l)*dum
            enddo 19
        endif
    enddo 21
enddo 22                     This is the end of the main loop over columns of the reduction.
do 24 l=n,1,-1                   It only remains to unscramble the solution in view
    if(indxr(l).ne.indxc(l))then     of the column interchanges. We do this by in-
        do 23 k=1,n                  terchanging pairs of columns in the reverse order
            dum=a(k,indxr(l))        that the permutation was built up.
            a(k,indxr(l))=a(k,indxc(l))
            a(k,indxc(l))=dum
        enddo 23
    endif
enddo 24
return                           And we are done.
END
```

Row versus Column Elimination Strategies

The above discussion can be amplified by a modest amount of formalism. Row operations on a matrix **A** correspond to pre- (that is, left-) multiplication by some simple

matrix $\mathbf{R}$. For example, the matrix $\mathbf{R}$ with components

$$R_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } i \neq 2, 4 \\ 1 & \text{if } i = 2, j = 4 \\ 1 & \text{if } i = 4, j = 2 \\ 0 & \text{otherwise} \end{cases} \tag{2.1.5}$$

effects the interchange of rows 2 and 4. Gauss-Jordan elimination by row operations alone (including the possibility of *partial* pivoting) consists of a series of such left-multiplications, yielding successively

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ (\cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{A}) \cdot \mathbf{x} &= \cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{b} \\ (\mathbf{1}) \cdot \mathbf{x} &= \cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{b} \\ \mathbf{x} &= \cdots \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{b} \end{aligned} \tag{2.1.6}$$

The key point is that since the $\mathbf{R}$'s build from right to left, the right-hand side is simply transformed at each stage from one vector to another.

Column operations, on the other hand, correspond to post-, or right-, multiplications by simple matrices, call them $\mathbf{C}$. The matrix in equation (2.1.5), if right-multiplied onto a matrix $\mathbf{A}$, will interchange $\mathbf{A}$'s second and fourth *columns*. Elimination by column operations involves (conceptually) inserting a column operator, *and also its inverse*, between the matrix $\mathbf{A}$ and the unknown vector $\mathbf{x}$:

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{A} \cdot \mathbf{C}_1 \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{A} \cdot \mathbf{C}_1 \cdot \mathbf{C}_2 \cdot \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \\ (\mathbf{A} \cdot \mathbf{C}_1 \cdot \mathbf{C}_2 \cdot \mathbf{C}_3 \cdots) \cdots \mathbf{C}_3^{-1} \cdot \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \\ (\mathbf{1}) \cdots \mathbf{C}_3^{-1} \cdot \mathbf{C}_2^{-1} \cdot \mathbf{C}_1^{-1} \cdot \mathbf{x} &= \mathbf{b} \end{aligned} \tag{2.1.7}$$

which (peeling of the $\mathbf{C}^{-1}$'s one at a time) implies a solution

$$\mathbf{x} = \mathbf{C}_1 \cdot \mathbf{C}_2 \cdot \mathbf{C}_3 \cdots \mathbf{b} \tag{2.1.8}$$

Notice the essential difference between equation (2.1.8) and equation (2.1.6). In the latter case, the $\mathbf{C}$'s must be applied to $\mathbf{b}$ in the *reverse order* from that in which they become known. That is, they must all be stored along the way. This requirement greatly reduces the usefulness of column operations, generally restricting them to simple permutations, for example in support of full pivoting.

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [1]

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Example 5.2, p. 282.

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Program B-2, p. 298.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.3–1.

2.2 Gaussian Elimination with Backsubstitution

The usefulness of Gaussian elimination with backsubstitution is primarily pedagogical. It stands between full elimination schemes such as Gauss-Jordan, and triangular decomposition schemes such as will be discussed in the next section. Gaussian elimination reduces a matrix not all the way to the identity matrix, but only halfway, to a matrix whose components on the diagonal and above (say) remain nontrivial. Let us now see what advantages accrue.

Suppose that in doing Gauss-Jordan elimination, as described in §2.1, we at each stage subtract away rows only *below* the then-current pivot element. When a_{22} is the pivot element, for example, we divide the second row by its value (as before), but now use the pivot row to zero only a_{32} and a_{42}, not a_{12} (see equation 2.1.1). Suppose, also, that we do only partial pivoting, never interchanging columns, so that the order of the unknowns never needs to be modified.

Then, when we have done this for all the pivots, we will be left with a reduced equation that looks like this (in the case of a single right-hand side vector):

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \tag{2.2.1}$$

Here the primes signify that the a's and b's do not have their original numerical values, but have been modified by all the row operations in the elimination to this point. The procedure up to this point is termed *Gaussian elimination*.

Backsubstitution

But how do we solve for the x's? The last x (x_4 in this example) is already isolated, namely

$$x_4 = b'_4/a'_{44} \tag{2.2.2}$$

With the last x known we can move to the penultimate x,

$$x_3 = \frac{1}{a'_{33}}[b'_3 - x_4 a'_{34}] \tag{2.2.3}$$

and then proceed with the x before that one. The typical step is

$$x_i = \frac{1}{a'_{ii}}\left[b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j\right] \tag{2.2.4}$$

The procedure defined by equation (2.2.4) is called *backsubstitution*. The combination of Gaussian elimination and backsubstitution yields a solution to the set of equations.

2.2 Gaussian Elimination with Backsubstitution

The usefulness of Gaussian elimination with backsubstitution is primarily pedagogical. It stands between full elimination schemes such as Gauss-Jordan, and triangular decomposition schemes such as will be discussed in the next section. Gaussian elimination reduces a matrix not all the way to the identity matrix, but only halfway, to a matrix whose components on the diagonal and above (say) remain nontrivial. Let us now see what advantages accrue.

Suppose that in doing Gauss-Jordan elimination, as described in §2.1, we at each stage subtract away rows only *below* the then-current pivot element. When a_{22} is the pivot element, for example, we divide the second row by its value (as before), but now use the pivot row to zero only a_{32} and a_{42}, not a_{12} (see equation 2.1.1). Suppose, also, that we do only partial pivoting, never interchanging columns, so that the order of the unknowns never needs to be modified.

Then, when we have done this for all the pivots, we will be left with a reduced equation that looks like this (in the case of a single right-hand side vector):

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \tag{2.2.1}$$

Here the primes signify that the a's and b's do not have their original numerical values, but have been modified by all the row operations in the elimination to this point. The procedure up to this point is termed *Gaussian elimination*.

Backsubstitution

But how do we solve for the x's? The last x (x_4 in this example) is already isolated, namely

$$x_4 = b'_4/a'_{44} \tag{2.2.2}$$

With the last x known we can move to the penultimate x,

$$x_3 = \frac{1}{a'_{33}}[b'_3 - x_4 a'_{34}] \tag{2.2.3}$$

and then proceed with the x before that one. The typical step is

$$x_i = \frac{1}{a'_{ii}}\left[b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j\right] \tag{2.2.4}$$

The procedure defined by equation (2.2.4) is called *backsubstitution*. The combination of Gaussian elimination and backsubstitution yields a solution to the set of equations.

The advantage of Gaussian elimination and backsubstitution over Gauss-Jordan elimination is simply that the former is faster in raw operations count: The innermost loops of Gauss-Jordan elimination, each containing one subtraction and one multiplication, are executed N^3 and N^2M times (where there are N equations and M unknowns). The corresponding loops in Gaussian elimination are executed only $\frac{1}{3}N^3$ times (only half the matrix is reduced, and the increasing numbers of predictable zeros reduce the count to one-third), and $\frac{1}{2}N^2M$ times, respectively. Each backsubstitution of a right-hand side is $\frac{1}{2}N^2$ executions of a similar loop (one multiplication plus one subtraction). For $M \ll N$ (only a few right-hand sides) Gaussian elimination thus has about a factor three advantage over Gauss-Jordan. (We could reduce this advantage to a factor 1.5 by *not* computing the inverse matrix as part of the Gauss-Jordan scheme.)

For computing the inverse matrix (which we can view as the case of $M = N$ right-hand sides, namely the N unit vectors which are the columns of the identity matrix), Gaussian elimination and backsubstitution at first glance require $\frac{1}{3}N^3$ (matrix reduction) $+\frac{1}{2}N^3$ (right-hand side manipulations) $+\frac{1}{2}N^3$ (N backsubstitutions) $= \frac{4}{3}N^3$ loop executions, which is more than the N^3 for Gauss-Jordan. However, the unit vectors are quite special in containing all zeros except for one element. If this is taken into account, the right-side manipulations can be reduced to only $\frac{1}{6}N^3$ loop executions, and, for matrix inversion, the two methods have identical efficiencies.

Both Gaussian elimination and Gauss-Jordan elimination share the disadvantage that all right-hand sides must be known in advance. The LU decomposition method in the next section does not share that deficiency, and also has an equally small operations count, both for solution with any number of right-hand sides, and for matrix inversion. For this reason we will not implement the method of Gaussian elimination as a routine.

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.3–1.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §2.1.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.2.1.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

2.3 LU Decomposition and Its Applications

Suppose we are able to write the matrix $\mathbf{A}$ as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{2.3.1}$$

where $\mathbf{L}$ is *lower triangular* (has elements only on the diagonal and below) and $\mathbf{U}$ is *upper triangular* (has elements only on the diagonal and above). For the case of

The advantage of Gaussian elimination and backsubstitution over Gauss-Jordan elimination is simply that the former is faster in raw operations count: The innermost loops of Gauss-Jordan elimination, each containing one subtraction and one multiplication, are executed N^3 and N^2M times (where there are N equations and M unknowns). The corresponding loops in Gaussian elimination are executed only $\frac{1}{3}N^3$ times (only half the matrix is reduced, and the increasing numbers of predictable zeros reduce the count to one-third), and $\frac{1}{2}N^2M$ times, respectively. Each backsubstitution of a right-hand side is $\frac{1}{2}N^2$ executions of a similar loop (one multiplication plus one subtraction). For $M \ll N$ (only a few right-hand sides) Gaussian elimination thus has about a factor three advantage over Gauss-Jordan. (We could reduce this advantage to a factor 1.5 by *not* computing the inverse matrix as part of the Gauss-Jordan scheme.)

For computing the inverse matrix (which we can view as the case of $M = N$ right-hand sides, namely the N unit vectors which are the columns of the identity matrix), Gaussian elimination and backsubstitution at first glance require $\frac{1}{3}N^3$ (matrix reduction) $+\frac{1}{2}N^3$ (right-hand side manipulations) $+\frac{1}{2}N^3$ (N backsubstitutions) $= \frac{4}{3}N^3$ loop executions, which is more than the N^3 for Gauss-Jordan. However, the unit vectors are quite special in containing all zeros except for one element. If this is taken into account, the right-side manipulations can be reduced to only $\frac{1}{6}N^3$ loop executions, and, for matrix inversion, the two methods have identical efficiencies.

Both Gaussian elimination and Gauss-Jordan elimination share the disadvantage that all right-hand sides must be known in advance. The LU decomposition method in the next section does not share that deficiency, and also has an equally small operations count, both for solution with any number of right-hand sides, and for matrix inversion. For this reason we will not implement the method of Gaussian elimination as a routine.

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.3–1.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §2.1.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.2.1.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

2.3 LU Decomposition and Its Applications

Suppose we are able to write the matrix $\mathbf{A}$ as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{2.3.1}$$

where $\mathbf{L}$ is *lower triangular* (has elements only on the diagonal and below) and $\mathbf{U}$ is *upper triangular* (has elements only on the diagonal and above). For the case of

a 4×4 matrix $\mathbf{A}$, for example, equation (2.3.1) would look like this:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{2.3.2}$$

We can use a decomposition such as (2.3.1) to solve the linear set

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{2.3.3}$$

by first solving for the vector $\mathbf{y}$ such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.3.4}$$

and then solving

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.3.5}$$

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equations is quite trivial, as we have already seen in §2.2 (equation 2.2.4). Thus, equation (2.3.4) can be solved by *forward substitution* as follows,

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}} \\ y_i &= \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \qquad i = 2, 3, \ldots, N \end{aligned} \tag{2.3.6}$$

while (2.3.5) can then be solved by *backsubstitution* exactly as in equations (2.2.2)–(2.2.4),

$$\begin{aligned} x_N &= \frac{y_N}{\beta_{NN}} \\ x_i &= \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right] \qquad i = N-1, N-2, \ldots, 1 \end{aligned} \tag{2.3.7}$$

Equations (2.3.6) and (2.3.7) total (for each right-hand side $\mathbf{b}$) N^2 executions of an inner loop containing one multiply and one add. If we have N right-hand sides which are the unit column vectors (which is the case when we are inverting a matrix), then taking into account the leading zeros reduces the total execution count of (2.3.6) from $\frac{1}{2}N^3$ to $\frac{1}{6}N^3$, while (2.3.7) is unchanged at $\frac{1}{2}N^3$.

Notice that, once we have the LU decomposition of $\mathbf{A}$, we can solve with as many right-hand sides as we then care to, one at a time. This is a distinct advantage over the methods of §2.1 and §2.2.

Performing the LU Decomposition

How then can we solve for **L** and **U**, given **A**? First, we write out the i,jth component of equation (2.3.1) or (2.3.2). That component always is a sum beginning with

$$\alpha_{i1}\beta_{1j} + \cdots = a_{ij}$$

The number of terms in the sum depends, however, on whether i or j is the smaller number. We have, in fact, the three cases,

$$i < j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.3.8}$$

$$i = j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.3.9}$$

$$i > j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.3.10}$$

Equations (2.3.8)–(2.3.10) total N^2 equations for the $N^2 + N$ unknown α's and β's (the diagonal being represented twice). Since the number of unknowns is greater than the number of equations, we are invited to specify N of the unknowns arbitrarily and then try to solve for the others. In fact, as we shall see, it is always possible to take

$$\alpha_{ii} \equiv 1 \qquad i = 1, \ldots, N \tag{2.3.11}$$

A surprising procedure, now, is *Crout's algorithm*, which quite trivially solves the set of $N^2 + N$ equations (2.3.8)–(2.3.11) for all the α's and β's by just arranging the equations in a certain order! That order is as follows:

- Set $\alpha_{ii} = 1$, $i = 1, \ldots, N$ (equation 2.3.11).
- For each $j = 1, 2, 3, \ldots, N$ do these two procedures: First, for $i = 1, 2, \ldots, j$, use (2.3.8), (2.3.9), and (2.3.11) to solve for β_{ij}, namely

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}. \tag{2.3.12}$$

  (When $i = 1$ in 2.3.12 the summation term is taken to mean zero.) Second, for $i = j+1, j+2, \ldots, N$ use (2.3.10) to solve for α_{ij}, namely

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right). \tag{2.3.13}$$

  Be sure to do both procedures before going on to the next j.

If you work through a few iterations of the above procedure, you will see that the α's and β's that occur on the right-hand side of equations (2.3.12) and (2.3.13) are already determined by the time they are needed. You will also see that every a_{ij} is used only once and never again. This means that the corresponding α_{ij} or β_{ij} can be stored in the location that the a used to occupy: the decomposition is "in place." [The diagonal unity elements α_{ii} (equation 2.3.11) are not stored at all.] In brief, Crout's method fills in the combined matrix of α's and β's,

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{2.3.14}$$

by columns from left to right, and within each column from top to bottom (see Figure 2.3.1).

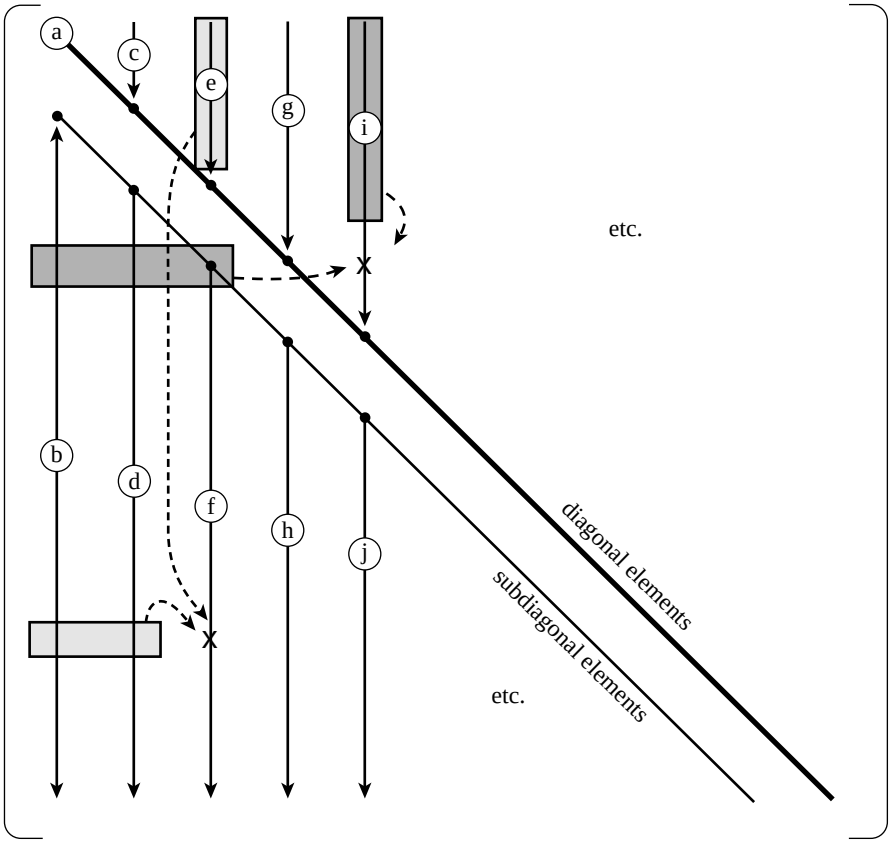


Figure 2.3.1. Crout's algorithm for *LU* decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lower case letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

What about pivoting? Pivoting (i.e., selection of a salubrious pivot element for the division in equation 2.3.13) is absolutely essential for the stability of Crout's method. Only partial pivoting (interchange of rows) can be implemented efficiently. However this is enough to make the method stable. This means, incidentally, that we don't actually decompose the matrix **A** into LU form, but rather we decompose a rowwise permutation of **A**. (If we keep track of what that permutation is, this decomposition is just as useful as the original one would have been.)

Pivoting is slightly subtle in Crout's algorithm. The key point to notice is that equation (2.3.12) in the case of $i = j$ (its final application) is *exactly the same* as equation (2.3.13) except for the division in the latter equation; in both cases the upper limit of the sum is $k = j - 1$ $(= i - 1)$. This means that we don't have to commit ourselves as to whether the diagonal element β_{jj} is the one that happens to fall on the diagonal in the first instance, or whether one of the (undivided) α_{ij}'s below it in the column, $i = j+1, \ldots, N$, is to be "promoted" to become the diagonal β. This can be decided after all the candidates in the column are in hand. As you should be able to guess by now, we will choose the largest one as the diagonal β (pivot element), then do all the divisions by that element *en masse*. This is *Crout's*

method with partial pivoting. Our implementation has one additional wrinkle: It initially finds the largest element in each row, and subsequently (when it is looking for the maximal pivot element) scales the comparison *as if* we had initially scaled all the equations to make their maximum coefficient equal to unity; this is the *implicit pivoting* mentioned in §2.1.

```
SUBROUTINE ludcmp(a,n,np,indx,d)
INTEGER n,np,indx(n),NMAX
REAL d,a(np,np),TINY
PARAMETER (NMAX=500,TINY=1.0e-20)  Largest expected n, and a small number.
```

Given a matrix `a(1:n,1:n)`, with physical dimension `np` by `np`, this routine replaces it by the LU decomposition of a rowwise permutation of itself. `a` and `n` are input. `a` is output, arranged as in equation (2.3.14) above; `indx(1:n)` is an output vector that records the row permutation effected by the partial pivoting; `d` is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with `lubksb` to solve linear equations or invert a matrix.

```
INTEGER i,imax,j,k
REAL aamax,dum,sum,vv(NMAX)          vv stores the implicit scaling of each row.
d=1.                                 No row interchanges yet.
do 12 i=1,n                          Loop over rows to get the implicit scaling informa-
    aamax=0.                             tion.
    do 11 j=1,n
        if (abs(a(i,j)).gt.aamax) aamax=abs(a(i,j))
    enddo 11
    if (aamax.eq.0.) pause 'singular matrix in ludcmp' No nonzero largest element.
    vv(i)=1./aamax                   Save the scaling.
enddo 12
do 19 j=1,n                          This is the loop over columns of Crout's method.
    do 14 i=1,j-1                    This is equation (2.3.12) except for i = j.
        sum=a(i,j)
        do 13 k=1,i-1
            sum=sum-a(i,k)*a(k,j)
        enddo 13
        a(i,j)=sum
    enddo 14
    aamax=0.                         Initialize for the search for largest pivot element.
    do 16 i=j,n                      This is i = j of equation (2.3.12) and i = j+1 ... N
        sum=a(i,j)                       of equation (2.3.13).
        do 15 k=1,j-1
            sum=sum-a(i,k)*a(k,j)
        enddo 15
        a(i,j)=sum
        dum=vv(i)*abs(sum)           Figure of merit for the pivot.
        if (dum.ge.aamax) then       Is it better than the best so far?
            imax=i
            aamax=dum
        endif
    enddo 16
    if (j.ne.imax)then               Do we need to interchange rows?
        do 17 k=1,n                  Yes, do so...
            dum=a(imax,k)
            a(imax,k)=a(j,k)
            a(j,k)=dum
        enddo 17
        d=-d                         ...and change the parity of d.
        vv(imax)=vv(j)               Also interchange the scale factor.
    endif
    indx(j)=imax
    if(a(j,j).eq.0.)a(j,j)=TINY
```

If the pivot element is zero the matrix is singular (at least to the precision of the algorithm). For some applications on singular matrices, it is desirable to substitute `TINY` for zero.

```
    if(j.ne.n)then                  Now, finally, divide by the pivot element.
        dum=1./a(j,j)
        do 18 i=j+1,n
            a(i,j)=a(i,j)*dum
        enddo 18
    endif
enddo 19                            Go back for the next column in the reduction.
return
END
```

Here is the routine for forward substitution and backsubstitution, implementing equations (2.3.6) and (2.3.7).

```
SUBROUTINE lubksb(a,n,np,indx,b)
INTEGER n,np,indx(n)
REAL a(np,np),b(n)
    Solves the set of n linear equations A · X = B. Here a is input, not as the matrix A but
    rather as its LU decomposition, determined by the routine ludcmp. indx is input as the
    permutation vector returned by ludcmp. b(1:n) is input as the right-hand side vector B,
    and returns with the solution vector X. a, n, np, and indx are not modified by this routine
    and can be left in place for successive calls with different right-hand sides b. This routine
    takes into account the possibility that b will begin with many zero elements, so it is efficient
    for use in matrix inversion.
INTEGER i,ii,j,ll
REAL sum
ii=0                            When ii is set to a positive value, it will become the in-
do 12 i=1,n                         dex of the first nonvanishing element of b. We now do
    ll=indx(i)                      the forward substitution, equation (2.3.6). The only new
    sum=b(ll)                       wrinkle is to unscramble the permutation as we go.
    b(ll)=b(i)
    if (ii.ne.0)then
        do 11 j=ii,i-1
            sum=sum-a(i,j)*b(j)
        enddo 11
    else if (sum.ne.0.) then
        ii=i                    A nonzero element was encountered, so from now on we will
    endif                           have to do the sums in the loop above.
    b(i)=sum
enddo 12
do 14 i=n,1,-1                  Now we do the backsubstitution, equation (2.3.7).
    sum=b(i)
    do 13 j=i+1,n
        sum=sum-a(i,j)*b(j)
    enddo 13
    b(i)=sum/a(i,i)             Store a component of the solution vector X.
enddo 14
return                          All done!
END
```

The LU decomposition in `ludcmp` requires about $\frac{1}{3}N^3$ executions of the inner loops (each with one multiply and one add). This is thus the operation count for solving one (or a few) right-hand sides, and is a factor of 3 better than the Gauss-Jordan routine `gaussj` which was given in §2.1, and a factor of 1.5 better than a Gauss-Jordan routine (not given) that does not compute the inverse matrix. For inverting a matrix, the total count (including the forward and backsubstitution as discussed following equation 2.3.7 above) is $(\frac{1}{3} + \frac{1}{6} + \frac{1}{2})N^3 = N^3$, the same as `gaussj`.

To summarize, this is the preferred way to solve the linear set of equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$:

```
call ludcmp(a,n,np,indx,d)
call lubksb(a,n,np,indx,b)
```

The answer **x** will be returned in `b`. Your original matrix **A** will have been destroyed.

If you subsequently want to solve a set of equations with the same **A** but a different right-hand side **b**, you repeat *only*

```
call lubksb(a,n,np,indx,b)
```

not, of course, with the original matrix **A**, but with `a` and `indx` as were already returned from `ludcmp`.

Inverse of a Matrix

Using the above LU decomposition and backsubstitution routines, it is completely straightforward to find the inverse of a matrix column by column.

```
INTEGER np,indx(np)
REAL a(np,np),y(np,np)
...
do 12 i=1,n                          Set up identity matrix.
    do 11 j=1,n
        y(i,j)=0.
    enddo 11
    y(i,i)=1.
enddo 12
call ludcmp(a,n,np,indx,d)           Decompose the matrix just once.
do 13 j=1,n                          Find inverse by columns.
    call lubksb(a,n,np,indx,y(1,j))
      Note that FORTRAN stores two-dimensional matrices by column, so y(1,j) is the
      address of the jth column of y.
enddo 13
```

The matrix `y` will now contain the inverse of the original matrix `a`, which will have been destroyed. Alternatively, there is nothing wrong with using a Gauss-Jordan routine like `gaussj` (§2.1) to invert a matrix in place, again destroying the original. Both methods have practically the same operations count.

Incidentally, if you ever have the need to compute $\mathbf{A}^{-1} \cdot \mathbf{B}$ from matrices **A** and **B**, you should LU decompose **A** and then backsubstitute with the columns of **B** instead of with the unit vectors that would give **A**'s inverse. This saves a whole matrix multiplication, and is also more accurate.

Determinant of a Matrix

The determinant of an LU decomposed matrix is just the product of the diagonal elements,

$$\det = \prod_{j=1}^{N} \beta_{jj} \tag{2.3.15}$$

We don't, recall, compute the decomposition of the original matrix, but rather a decomposition of a rowwise permutation of it. Luckily, we have kept track of whether the number of row interchanges was even or odd, so we just preface the product by the corresponding sign. (You now finally know the purpose of returning `d` in the routine `ludcmp`.)

Calculation of a determinant thus requires one call to `ludcmp`, with *no* subsequent backsubstitutions by `lubksb`.

```
INTEGER np,indx(np)
REAL a(np,np)
...
call ludcmp(a,n,np,indx,d)          This returns d as ±1.
do 11 j=1,n
    d=d*a(j,j)
enddo 11
```

The variable `d` now contains the determinant of the original matrix `a`, which will have been destroyed.

For a matrix of any substantial size, it is quite likely that the determinant will overflow or underflow your computer's floating-point dynamic range. In this case you can modify the loop of the above fragment and (e.g.) divide by powers of ten, to keep track of the scale separately, or (e.g.) accumulate the sum of logarithms of the absolute values of the factors and the sign separately.

Complex Systems of Equations

If your matrix $\mathbf{A}$ is real, but the right-hand side vector is complex, say $\mathbf{b} + i\mathbf{d}$, then (i) LU decompose $\mathbf{A}$ in the usual way, (ii) backsubstitute $\mathbf{b}$ to get the real part of the solution vector, and (iii) backsubstitute $\mathbf{d}$ to get the imaginary part of the solution vector.

If the matrix itself is complex, so that you want to solve the system

$$(\mathbf{A} + i\mathbf{C}) \cdot (\mathbf{x} + i\mathbf{y}) = (\mathbf{b} + i\mathbf{d}) \tag{2.3.16}$$

then there are two possible ways to proceed. The best way is to rewrite `ludcmp` and `lubksb` as complex routines. Complex modulus substitutes for absolute value in the construction of the scaling vector `vv` and in the search for the largest pivot elements. Everything else goes through in the obvious way, with complex arithmetic used as needed.

A quick-and-dirty way to solve complex systems is to take the real and imaginary parts of (2.3.16), giving

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} - \mathbf{C} \cdot \mathbf{y} &= \mathbf{b} \\ \mathbf{C} \cdot \mathbf{x} + \mathbf{A} \cdot \mathbf{y} &= \mathbf{d} \end{aligned} \tag{2.3.17}$$

which can be written as a $2N \times 2N$ set of *real* equations,

$$\begin{pmatrix} \mathbf{A} & -\mathbf{C} \\ \mathbf{C} & \mathbf{A} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{d} \end{pmatrix} \tag{2.3.18}$$

and then solved with `ludcmp` and `lubksb` in their present forms. This scheme is a factor of 2 inefficient in storage, since **A** and **C** are stored twice. It is also a factor of 2 inefficient in time, since the complex multiplies in a complexified version of the routines would each use 4 real multiplies, while the solution of a $2N \times 2N$ problem involves 8 times the work of an $N \times N$ one. If you can tolerate these factor-of-two inefficiencies, then equation (2.3.18) is an easy way to proceed.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §3.3, and p. 50.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 9, 16, and 18.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.11.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

2.4 Tridiagonal and Band Diagonal Systems of Equations

The special case of a system of linear equations that is *tridiagonal*, that is, has nonzero elements only on the diagonal plus or minus one column, is one that occurs frequently. Also common are systems that are *band diagonal*, with nonzero elements only along a few diagonal lines adjacent to the main diagonal (above and below).

For tridiagonal sets, the procedures of LU decomposition, forward- and back-substitution each take only $O(N)$ operations, and the whole solution can be encoded very concisely. The resulting routine `tridag` is one that we will use in later chapters.

Naturally, one does not reserve storage for the full $N \times N$ matrix, but only for the nonzero components, stored as three vectors. The set of equations to be solved is

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \cdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.4.1}$$

Notice that a_1 and c_N are undefined and are not referenced by the routine that follows.

and then solved with `ludcmp` and `lubksb` in their present forms. This scheme is a factor of 2 inefficient in storage, since **A** and **C** are stored twice. It is also a factor of 2 inefficient in time, since the complex multiplies in a complexified version of the routines would each use 4 real multiplies, while the solution of a $2N \times 2N$ problem involves 8 times the work of an $N \times N$ one. If you can tolerate these factor-of-two inefficiencies, then equation (2.3.18) is an easy way to proceed.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §3.3, and p. 50.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 9, 16, and 18.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.11.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

2.4 Tridiagonal and Band Diagonal Systems of Equations

The special case of a system of linear equations that is *tridiagonal*, that is, has nonzero elements only on the diagonal plus or minus one column, is one that occurs frequently. Also common are systems that are *band diagonal*, with nonzero elements only along a few diagonal lines adjacent to the main diagonal (above and below).

For tridiagonal sets, the procedures of LU decomposition, forward- and back-substitution each take only $O(N)$ operations, and the whole solution can be encoded very concisely. The resulting routine `tridag` is one that we will use in later chapters.

Naturally, one does not reserve storage for the full $N \times N$ matrix, but only for the nonzero components, stored as three vectors. The set of equations to be solved is

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \cdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.4.1}$$

Notice that a_1 and c_N are undefined and are not referenced by the routine that follows.

```
SUBROUTINE tridag(a,b,c,r,u,n)
INTEGER n,NMAX
REAL a(n),b(n),c(n),r(n),u(n)
PARAMETER (NMAX=500)
    Solves for a vector u(1:n) of length n the tridiagonal linear set given by equation (2.4.1).
    a(1:n), b(1:n), c(1:n), and r(1:n) are input vectors and are not modified.
    Parameter: NMAX is the maximum expected value of n.
INTEGER j
REAL bet,gam(NMAX)                          One vector of workspace, gam is needed.
if(b(1).eq.0.)pause 'tridag: rewrite equations'
  If this happens then you should rewrite your equations as a set of order N − 1, with u2
  trivially eliminated.
bet=b(1)
u(1)=r(1)/bet
do 11 j=2,n                                 Decomposition and forward substitution.
    gam(j)=c(j-1)/bet
    bet=b(j)-a(j)*gam(j)
    if(bet.eq.0.)pause 'tridag failed'      Algorithm fails; see below.
    u(j)=(r(j)-a(j)*u(j-1))/bet
enddo 11
do 12 j=n-1,1,-1                            Backsubstitution.
    u(j)=u(j)-gam(j+1)*u(j+1)
enddo 12
return
END
```

There is no pivoting in `tridag`. It is for this reason that `tridag` can fail (`pause`) even when the underlying matrix is nonsingular: A zero pivot can be encountered even for a nonsingular matrix. In practice, this is not something to lose sleep about. The kinds of problems that lead to tridiagonal linear sets usually have additional properties which guarantee that the algorithm in `tridag` will succeed. For example, if

$$|b_j| > |a_j| + |c_j| \qquad j = 1, \ldots, N \tag{2.4.2}$$

(called *diagonal dominance*) then it can be shown that the algorithm cannot encounter a zero pivot.

It is possible to construct special examples in which the lack of pivoting in the algorithm causes numerical instability. In practice, however, such instability is almost never encountered — unlike the general matrix problem where pivoting is essential.

The tridiagonal algorithm is the rare case of an algorithm that, in practice, is more robust than theory says it should be. Of course, should you ever encounter a problem for which `tridag` fails, you can instead use the more general method for band diagonal systems, now described (routines `bandec` and `banbks`).

Some other matrix forms consisting of tridiagonal with a small number of additional elements (e.g., upper right and lower left corners) also allow rapid solution; see §2.7.

Band Diagonal Systems

Where tridiagonal systems have nonzero elements only on the diagonal plus or minus one, band diagonal systems are slightly more general and have (say) $m_1 \geq 0$ nonzero elements immediately to the left of (below) the diagonal and $m_2 \geq 0$ nonzero elements immediately to its right (above it). Of course, this is only a useful classification if m_1 and m_2 are both $\ll N$. In that case, the solution of the linear system by LU decomposition can be accomplished much faster, and in much less storage, than for the general $N \times N$ case.

The precise definition of a band diagonal matrix with elements a_{ij} is that

$$a_{ij} = 0 \quad \text{when} \quad j > i + m_2 \quad \text{or} \quad i > j + m_1 \tag{2.4.3}$$

Band diagonal matrices are stored and manipulated in a so-called compact form, which results if the matrix is tilted 45° clockwise, so that its nonzero elements lie in a long, narrow matrix with $m_1 + 1 + m_2$ columns and N rows. This is best illustrated by an example: The band diagonal matrix

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 0 & 0 & 0 \\ 9 & 2 & 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 9 & 0 & 0 \\ 0 & 0 & 7 & 9 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 8 & 4 & 6 \\ 0 & 0 & 0 & 0 & 2 & 4 & 4 \end{pmatrix} \tag{2.4.4}$$

which has $N = 7$, $m_1 = 2$, and $m_2 = 1$, is stored compactly as the 7×4 matrix,

$$\begin{pmatrix} x & x & 3 & 1 \\ x & 4 & 1 & 5 \\ 9 & 2 & 6 & 5 \\ 3 & 5 & 8 & 9 \\ 7 & 9 & 3 & 2 \\ 3 & 8 & 4 & 6 \\ 2 & 4 & 4 & x \end{pmatrix} \tag{2.4.5}$$

Here x denotes elements that are wasted space in the compact format; these will not be referenced by any manipulations and can have arbitrary values. Notice that the diagonal of the original matrix appears in column $m_1 + 1$, with subdiagonal elements to its left, superdiagonal elements to its right.

The simplest manipulation of a band diagonal matrix, stored compactly, is to multiply it by a vector to its right. Although this is algorithmically trivial, you might want to study the following routine carefully, as an example of how to pull nonzero elements a_{ij} out of the compact storage format in an orderly fashion. Notice that, as always, the logical and physical dimensions of a two-dimensional array can be different. Our convention is to pass N, m_1, m_2, and the *physical* dimensions $\texttt{np} \geq N$ and $\texttt{mp} \geq m_1 + 1 + m_2$.

```
SUBROUTINE banmul(a,n,m1,m2,np,mp,x,b)
INTEGER m1,m2,mp,n,np
REAL a(np,mp),b(n),x(n)
```

Matrix multiply $\mathbf{b} = \mathbf{A} \cdot \mathbf{x}$, where $\mathbf{A}$ is band diagonal with `m1` rows below the diagonal and `m2` rows above. The input vector $\mathbf{x}$ and output vector $\mathbf{b}$ are stored as `x(1:n)` and `b(1:n)`, respectively. The array `a(1:n,1:m1+m2+1)` stores $\mathbf{A}$ as follows: The diagonal elements are in `a(1:n,m1+1)`. Subdiagonal elements are in `a(`j`:n,1:m1)` (with $j > 1$ appropriate to the number of elements on each subdiagonal). Superdiagonal elements are in `a(1:`j`,m1+2:m1+m2+1)` with $j <$ `n` appropriate to the number of elements on each superdiagonal.

```
INTEGER i,j,k
do 12 i=1,n
    b(i)=0.
    k=i-m1-1
    do 11 j=max(1,1-k),min(m1+m2+1,n-k)
        b(i)=b(i)+a(i,j)*x(j+k)
    enddo 11
enddo 12
return
END
```

It is not possible to store the LU decomposition of a band diagonal matrix **A** quite as compactly as the compact form of **A** itself. The decomposition (essentially by Crout's method, see §2.3) produces additional nonzero "fill-ins." One straightforward storage scheme is to return the upper triangular factor (U) in the same space that **A** previously occupied, and to return the lower triangular factor (L) in a separate compact matrix of size $N \times m_1$. The diagonal elements of U (whose product, times d $= \pm 1$, gives the determinant) are returned in the first column of **A**'s storage space.

The following routine, bandec, is the band-diagonal analog of ludcmp in §2.3:

```
SUBROUTINE bandec(a,n,m1,m2,np,mp,al,mpl,indx,d)
INTEGER m1,m2,mp,mpl,n,np,indx(n)
REAL d,a(np,mp),al(np,mpl),TINY
PARAMETER (TINY=1.e-20)
```

Given an n × n band diagonal matrix **A** with m1 subdiagonal rows and m2 superdiagonal rows, compactly stored in the array a(1:n,1:m1+m2+1) as described in the comment for routine banmul, this routine constructs an LU decomposition of a rowwise permutation of **A**. The upper triangular matrix replaces a, while the lower triangular matrix is returned in al(1:n,1:m1). indx(1:n) is an output vector which records the row permutation effected by the partial pivoting; d is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with banbks to solve band-diagonal sets of equations.

```
INTEGER i,j,k,l,mm
REAL dum
mm=m1+m2+1
if(mm.gt.mp.or.m1.gt.mpl.or.n.gt.np) pause 'bad args in bandec'
l=m1
do 13 i=1,m1                    Rearrange the storage a bit.
    do 11 j=m1+2-i,mm
        a(i,j-l)=a(i,j)
    enddo 11
    l=l-1
    do 12 j=mm-l,mm
        a(i,j)=0.
    enddo 12
enddo 13
d=1.
l=m1
do 18 k=1,n                     For each row...
    dum=a(k,1)
    i=k
    if(l.lt.n)l=l+1
    do 14 j=k+1,l               Find the pivot element.
        if(abs(a(j,1)).gt.abs(dum))then
            dum=a(j,1)
            i=j
        endif
    enddo 14
    indx(k)=i
    if(dum.eq.0.) a(k,1)=TINY
      Matrix is algorithmically singular, but proceed anyway with TINY pivot (desirable in some applications).
    if(i.ne.k)then              Interchange rows.
        d=-d
        do 15 j=1,mm
            dum=a(k,j)
            a(k,j)=a(i,j)
            a(i,j)=dum
        enddo 15
    endif
    do 17 i=k+1,l               Do the elimination.
        dum=a(i,1)/a(k,1)
        al(k,i-k)=dum
        do 16 j=2,mm
```

```
            a(i,j-1)=a(i,j)-dum*a(k,j)
        enddo 16
        a(i,mm)=0.
    enddo 17
enddo 18
return
END
```

Some pivoting is possible within the storage limitations of `bandec`, and the above routine does take advantage of the opportunity. In general, when `TINY` is returned as a diagonal element of U, then the original matrix (perhaps as modified by roundoff error) is in fact singular. In this regard, `bandec` is somewhat more robust than `tridag` above, which can fail algorithmically even for nonsingular matrices; `bandec` is thus also useful (with $m_1 = m_2 = 1$) for some ill-behaved tridiagonal systems.

Once the matrix $\mathbf{A}$ has been decomposed, any number of right-hand sides can be solved in turn by repeated calls to `banbks`, the backsubstitution routine whose analog in §2.3 is `lubksb`.

```
SUBROUTINE banbks(a,n,m1,m2,np,mp,al,mpl,indx,b)
INTEGER m1,m2,mp,mpl,n,np,indx(n)
REAL a(np,mp),al(np,mpl),b(n)
    Given the arrays a, al, and indx as returned from bandec, and given a right-hand side
    vector b(1:n), solves the band diagonal linear equations A · x = b. The solution vector x
    overwrites b(1:n). The other input arrays are not modified, and can be left in place for
    successive calls with different right-hand sides.
INTEGER i,k,l,mm
REAL dum
mm=m1+m2+1
if(mm.gt.mp.or.m1.gt.mpl.or.n.gt.np) pause 'bad args in banbks'
l=m1
do 12 k=1,n                         Forward substitution, unscrambling the permuted rows as we
    i=indx(k)                           go.
    if(i.ne.k)then
        dum=b(k)
        b(k)=b(i)
        b(i)=dum
    endif
    if(l.lt.n)l=l+1
    do 11 i=k+1,l
        b(i)=b(i)-al(k,i-k)*b(k)
    enddo 11
enddo 12
l=1
do 14 i=n,1,-1                      Backsubstitution.
    dum=b(i)
    do 13 k=2,l
        dum=dum-a(i,k)*b(k+i-1)
    enddo 13
    b(i)=dum/a(i,1)
    if(l.lt.mm) l=l+1
enddo 14
return
END
```

The routines `bandec` and `banbks` are based on the Handbook routines *bandet1* and *bansol1* in [1].

CITED REFERENCES AND FURTHER READING:

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell), p. 74.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Example 5.4.3, p. 166.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.11.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/6. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.3.

2.5 Iterative Improvement of a Solution to Linear Equations

Obviously it is not easy to obtain greater precision for the solution of a linear set than the precision of your computer's floating-point word. Unfortunately, for large sets of linear equations, it is not always easy to obtain precision equal to, or even comparable to, the computer's limit. In direct methods of solution, roundoff errors accumulate, and they are magnified to the extent that your matrix is close to singular. You can easily lose two or three significant figures for matrices which (you thought) were *far* from singular.

If this happens to you, there is a neat trick to restore the full machine precision, called *iterative improvement* of the solution. The theory is very straightforward (see Figure 2.5.1): Suppose that a vector $\mathbf{x}$ is the exact solution of the linear set

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.5.1}$$

You don't, however, know $\mathbf{x}$. You only know some slightly wrong solution $\mathbf{x} + \delta\mathbf{x}$, where $\delta\mathbf{x}$ is the unknown error. When multiplied by the matrix $\mathbf{A}$, your slightly wrong solution gives a product slightly discrepant from the desired right-hand side $\mathbf{b}$, namely

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{2.5.2}$$

Subtracting (2.5.1) from (2.5.2) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b} \tag{2.5.3}$$

But (2.5.2) can also be solved, trivially, for $\delta\mathbf{b}$. Substituting this into (2.5.3) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) - \mathbf{b} \tag{2.5.4}$$

In this equation, the whole right-hand side is known, since $\mathbf{x} + \delta\mathbf{x}$ is the wrong solution that you want to improve. It is essential to calculate the right-hand side in double precision, since there will be a lot of cancellation in the subtraction of $\mathbf{b}$. Then, we need only solve (2.5.4) for the error $\delta\mathbf{x}$, then subtract this from the wrong solution to get an improved solution.

An important extra benefit occurs if we obtained the original solution by LU decomposition. In this case we already have the LU decomposed form of $\mathbf{A}$, and all we need do to solve (2.5.4) is compute the right-hand side and backsubstitute!

The code to do all this is concise and straightforward:

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Example 5.4.3, p. 166.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.11.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/6. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.3.

2.5 Iterative Improvement of a Solution to Linear Equations

Obviously it is not easy to obtain greater precision for the solution of a linear set than the precision of your computer's floating-point word. Unfortunately, for large sets of linear equations, it is not always easy to obtain precision equal to, or even comparable to, the computer's limit. In direct methods of solution, roundoff errors accumulate, and they are magnified to the extent that your matrix is close to singular. You can easily lose two or three significant figures for matrices which (you thought) were *far* from singular.

If this happens to you, there is a neat trick to restore the full machine precision, called *iterative improvement* of the solution. The theory is very straightforward (see Figure 2.5.1): Suppose that a vector $\mathbf{x}$ is the exact solution of the linear set

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.5.1}$$

You don't, however, know $\mathbf{x}$. You only know some slightly wrong solution $\mathbf{x} + \delta\mathbf{x}$, where $\delta\mathbf{x}$ is the unknown error. When multiplied by the matrix $\mathbf{A}$, your slightly wrong solution gives a product slightly discrepant from the desired right-hand side $\mathbf{b}$, namely

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{2.5.2}$$

Subtracting (2.5.1) from (2.5.2) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b} \tag{2.5.3}$$

But (2.5.2) can also be solved, trivially, for $\delta\mathbf{b}$. Substituting this into (2.5.3) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) - \mathbf{b} \tag{2.5.4}$$

In this equation, the whole right-hand side is known, since $\mathbf{x} + \delta\mathbf{x}$ is the wrong solution that you want to improve. It is essential to calculate the right-hand side in double precision, since there will be a lot of cancellation in the subtraction of $\mathbf{b}$. Then, we need only solve (2.5.4) for the error $\delta\mathbf{x}$, then subtract this from the wrong solution to get an improved solution.

An important extra benefit occurs if we obtained the original solution by LU decomposition. In this case we already have the LU decomposed form of $\mathbf{A}$, and all we need do to solve (2.5.4) is compute the right-hand side and backsubstitute!

The code to do all this is concise and straightforward:

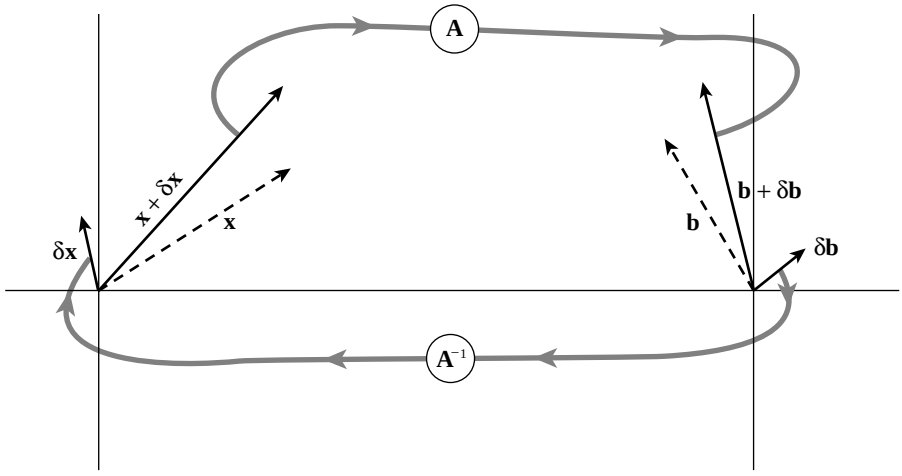


Figure 2.5.1. Iterative improvement of the solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. The first guess $\mathbf{x} + \delta\mathbf{x}$ is multiplied by $\mathbf{A}$ to produce $\mathbf{b} + \delta\mathbf{b}$. The known vector $\mathbf{b}$ is subtracted, giving $\delta\mathbf{b}$. The linear set with this right-hand side is inverted, giving $\delta\mathbf{x}$. This is subtracted from the first guess giving an improved solution $\mathbf{x}$.

```
      SUBROUTINE mprove(a,alud,n,np,indx,b,x)
      INTEGER n,np,indx(n),NMAX
      REAL a(np,np),alud(np,np),b(n),x(n)
      PARAMETER (NMAX=500)                    Maximum anticipated value of n.
C     USES lubksb
        Improves a solution vector x(1:n) of the linear set of equations A · X = B. The matrix
        a(1:n,1:n), and the vectors b(1:n) and x(1:n) are input, as is the dimension n. Also
        input is alud, the LU decomposition of a as returned by ludcmp, and the vector indx also
        returned by that routine. On output, only x(1:n) is modified, to an improved set of values.
      INTEGER i,j
      REAL r(NMAX)
      DOUBLE PRECISION sdp
      do 12 i=1,n                             Calculate the right-hand side, accumulating the resid-
        sdp=-b(i)                                 ual in double precision.
        do 11 j=1,n
          sdp=sdp+dble(a(i,j))*dble(x(j))
        enddo 11
        r(i)=sdp
      enddo 12
      call lubksb(alud,n,np,indx,r)           Solve for the error term,
      do 13 i=1,n                             and subtract it from the old solution.
        x(i)=x(i)-r(i)
      enddo 13
      return
      END
```

You should note that the routine `ludcmp` in §2.3 destroys the input matrix as it LU decomposes it. Since iterative improvement requires *both* the original matrix and its LU decomposition, you will need to copy $\mathbf{A}$ before calling `ludcmp`. Likewise `lubksb` destroys $\mathbf{b}$ in obtaining $\mathbf{x}$, so make a copy of $\mathbf{b}$ also. If you don't mind this extra storage, iterative improvement is *highly* recommended: It is a process of order only N^2 operations (multiply vector by matrix, and backsubstitute — see discussion following equation 2.3.7); it never hurts; and it can really give you your money's worth if it saves an otherwise ruined solution on which you have already spent of order N^3 operations.

You can call `mprove` several times in succession if you want. Unless you are starting quite far from the true solution, one call is generally enough; but a second call to verify convergence can be reassuring.

More on Iterative Improvement

It is illuminating (and will be useful later in the book) to give a somewhat more solid analytical foundation for equation (2.5.4), and also to give some additional results. Implicit in the previous discussion was the notion that the solution vector $\mathbf{x} + \delta\mathbf{x}$ has an error term; but we neglected the fact that the LU decomposition of $\mathbf{A}$ is itself not exact.

A different analytical approach starts with some matrix $\mathbf{B}_0$ that is assumed to be an *approximate* inverse of the matrix $\mathbf{A}$, so that $\mathbf{B}_0 \cdot \mathbf{A}$ is approximately the identity matrix $\mathbf{1}$. Define the *residual matrix* $\mathbf{R}$ of $\mathbf{B}_0$ as

$$\mathbf{R} \equiv \mathbf{1} - \mathbf{B}_0 \cdot \mathbf{A} \tag{2.5.5}$$

which is supposed to be "small" (we will be more precise below). Note that therefore

$$\mathbf{B}_0 \cdot \mathbf{A} = \mathbf{1} - \mathbf{R} \tag{2.5.6}$$

Next consider the following formal manipulation:

$$\begin{aligned} \mathbf{A}^{-1} &= \mathbf{A}^{-1} \cdot (\mathbf{B}_0^{-1} \cdot \mathbf{B}_0) = (\mathbf{A}^{-1} \cdot \mathbf{B}_0^{-1}) \cdot \mathbf{B}_0 = (\mathbf{B}_0 \cdot \mathbf{A})^{-1} \cdot \mathbf{B}_0 \\ &= (\mathbf{1} - \mathbf{R})^{-1} \cdot \mathbf{B}_0 = (\mathbf{1} + \mathbf{R} + \mathbf{R}^2 + \mathbf{R}^3 + \cdots) \cdot \mathbf{B}_0 \end{aligned} \tag{2.5.7}$$

We can define the nth partial sum of the last expression by

$$\mathbf{B}_n \equiv (\mathbf{1} + \mathbf{R} + \cdots + \mathbf{R}^n) \cdot \mathbf{B}_0 \tag{2.5.8}$$

so that $\mathbf{B}_\infty \rightarrow \mathbf{A}^{-1}$, if the limit exists.

It now is straightforward to verify that equation (2.5.8) satisfies some interesting recurrence relations. As regards solving $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{x}$ and $\mathbf{b}$ are vectors, define

$$\mathbf{x}_n \equiv \mathbf{B}_n \cdot \mathbf{b} \tag{2.5.9}$$

Then it is easy to show that

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{B}_0 \cdot (\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_n) \tag{2.5.10}$$

This is immediately recognizable as equation (2.5.4), with $-\delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n$, and with $\mathbf{B}_0$ taking the role of $\mathbf{A}^{-1}$. We see, therefore, that equation (2.5.4) does not require that the LU decompositon of $\mathbf{A}$ be exact, but only that the implied residual $\mathbf{R}$ be small. In rough terms, if the residual is smaller than the square root of your computer's roundoff error, then after one application of equation (2.5.10) (that is, going from $\mathbf{x}_0 \equiv \mathbf{B}_0 \cdot \mathbf{b}$ to $\mathbf{x}_1$) the first neglected term, of order $\mathbf{R}^2$, will be smaller than the roundoff error. Equation (2.5.10), like equation (2.5.4), moreover, can be applied more than once, since it uses only $\mathbf{B}_0$, and not any of the higher $\mathbf{B}$'s.

A much more surprising recurrence which follows from equation (2.5.8) is one that more than *doubles* the order n at each stage:

$$\mathbf{B}_{2n+1} = 2\mathbf{B}_n - \mathbf{B}_n \cdot \mathbf{A} \cdot \mathbf{B}_n \qquad n = 0, 1, 3, 7, \ldots \tag{2.5.11}$$

Repeated application of equation (2.5.11), from a suitable starting matrix $\mathbf{B}_0$, converges *quadratically* to the unknown inverse matrix $\mathbf{A}^{-1}$ (see §9.4 for the definition of "quadratically"). Equation (2.5.11) goes by various names, including *Schultz's Method* and *Hotelling's Method*; see Pan and Reif [1] for references. In fact, equation (2.5.11) is simply the iterative Newton-Raphson method of root-finding (§9.4) applied to matrix inversion.

Before you get too excited about equation (2.5.11), however, you should notice that it involves two full matrix multiplications at each iteration. Each matrix multiplication involves N^3 adds and multiplies. But we already saw in §§2.1–2.3 that direct inversion of $\mathbf{A}$ requires only N^3 adds and N^3 multiplies *in toto*. Equation (2.5.11) is therefore practical only when special circumstances allow it to be evaluated much more rapidly than is the case for general matrices. We will meet such circumstances later, in §13.10.

In the spirit of delayed gratification, let us nevertheless pursue the two related issues: When does the series in equation (2.5.7) converge; and what is a suitable initial guess $\mathbf{B}_0$ (if, for example, an initial LU decomposition is not feasible)?

We can define the norm of a matrix as the largest amplification of length that it is able to induce on a vector,

$$\|\mathbf{R}\| \equiv \max_{\mathbf{v}\neq 0} \frac{|\mathbf{R}\cdot\mathbf{v}|}{|\mathbf{v}|} \tag{2.5.12}$$

If we let equation (2.5.7) act on some arbitrary right-hand side $\mathbf{b}$, as one wants a matrix inverse to do, it is obvious that a sufficient condition for convergence is

$$\|\mathbf{R}\| < 1 \tag{2.5.13}$$

Pan and Reif [1] point out that a suitable initial guess for $\mathbf{B}_0$ is any sufficiently small constant ϵ times the matrix transpose of $\mathbf{A}$, that is,

$$\mathbf{B}_0 = \epsilon\mathbf{A}^T \qquad \text{or} \qquad \mathbf{R} = \mathbf{1} - \epsilon\mathbf{A}^T\cdot\mathbf{A} \tag{2.5.14}$$

To see why this is so involves concepts from Chapter 11; we give here only the briefest sketch: $\mathbf{A}^T\cdot\mathbf{A}$ is a symmetric, positive definite matrix, so it has real, positive eigenvalues. In its diagonal representation, $\mathbf{R}$ takes the form

$$\mathbf{R} = \mathrm{diag}(1-\epsilon\lambda_1, 1-\epsilon\lambda_2, \ldots, 1-\epsilon\lambda_N) \tag{2.5.15}$$

where all the λ_i's are positive. Evidently any ϵ satisfying $0 < \epsilon < 2/(\max_i \lambda_i)$ will give $\|\mathbf{R}\| < 1$. It is not difficult to show that the optimal choice for ϵ, giving the most rapid convergence for equation (2.5.11), is

$$\epsilon = 2/(\max_i \lambda_i + \min_i \lambda_i) \tag{2.5.16}$$

Rarely does one know the eigenvalues of $\mathbf{A}^T\cdot\mathbf{A}$ in equation (2.5.16). Pan and Reif derive several interesting bounds, which are computable directly from $\mathbf{A}$. The following choices guarantee the convergence of $\mathbf{B}_n$ as $n\to\infty$,

$$\epsilon \le 1\Big/\sum_{j,k} a_{jk}^2 \qquad \text{or} \qquad \epsilon \le 1\Big/\left(\max_i \sum_j |a_{ij}| \times \max_j \sum_i |a_{ij}|\right) \tag{2.5.17}$$

The latter expression is truly a remarkable formula, which Pan and Reif derive by noting that the vector norm in equation (2.5.12) need not be the usual L_2 norm, but can instead be either the L_∞ (max) norm, or the L_1 (absolute value) norm. See their work for details.

Another approach, with which we have had some success, is to estimate the largest eigenvalue statistically, by calculating $s_i \equiv |\mathbf{A}\cdot\mathbf{v}_i|^2$ for several unit vector $\mathbf{v}_i$'s with randomly chosen directions in N-space. The largest eigenvalue λ can then be bounded by the maximum of $2\max s_i$ and $2N\mathrm{Var}(s_i)/\mu(s_i)$, where Var and μ denote the sample variance and mean, respectively.

CITED REFERENCES AND FURTHER READING:

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.3.4, p. 55.

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), p. 74.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.5.6, p. 183.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.5, p. 437.

Pan, V., and Reif, J. 1985, in Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (New York: Association for Computing Machinery). [1]

2.6 Singular Value Decomposition

There exists a very powerful set of techniques for dealing with sets of equations or matrices that are either singular or else numerically very close to singular. In many cases where Gaussian elimination and LU decomposition fail to give satisfactory results, this set of techniques, known as *singular value decomposition*, or *SVD*, will diagnose for you precisely what the problem is. In some cases, SVD will not only diagnose the problem, it will also solve it, in the sense of giving you a useful numerical answer, although, as we shall see, not necessarily "the" answer that you thought you should get.

SVD is also the method of choice for solving most *linear least-squares* problems. We will outline the relevant theory in this section, but defer detailed discussion of the use of SVD in this application to Chapter 15, whose subject is the parametric modeling of data.

SVD methods are based on the following theorem of linear algebra, whose proof is beyond our scope: Any $M \times N$ matrix $\mathbf{A}$ whose number of rows M is greater than or equal to its number of columns N, can be written as the product of an $M \times N$ column-orthogonal matrix $\mathbf{U}$, an $N \times N$ diagonal matrix $\mathbf{W}$ with positive or zero elements (the *singular values*), and the transpose of an $N \times N$ orthogonal matrix $\mathbf{V}$. The various shapes of these matrices will be made clearer by the following tableau:

$$\begin{pmatrix} & & \\ & \mathbf{A} & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{U} & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} \cdot \begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{pmatrix} \tag{2.6.1}$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ are each orthogonal in the sense that their columns are orthonormal,

$$\sum_{i=1}^{M} U_{ik} U_{in} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.6.2}$$

$$\sum_{j=1}^{N} V_{jk} V_{jn} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.6.3}$$

2.6 Singular Value Decomposition

There exists a very powerful set of techniques for dealing with sets of equations or matrices that are either singular or else numerically very close to singular. In many cases where Gaussian elimination and LU decomposition fail to give satisfactory results, this set of techniques, known as *singular value decomposition*, or *SVD*, will diagnose for you precisely what the problem is. In some cases, SVD will not only diagnose the problem, it will also solve it, in the sense of giving you a useful numerical answer, although, as we shall see, not necessarily "the" answer that you thought you should get.

SVD is also the method of choice for solving most *linear least-squares* problems. We will outline the relevant theory in this section, but defer detailed discussion of the use of SVD in this application to Chapter 15, whose subject is the parametric modeling of data.

SVD methods are based on the following theorem of linear algebra, whose proof is beyond our scope: Any $M \times N$ matrix $\mathbf{A}$ whose number of rows M is greater than or equal to its number of columns N, can be written as the product of an $M \times N$ column-orthogonal matrix $\mathbf{U}$, an $N \times N$ diagonal matrix $\mathbf{W}$ with positive or zero elements (the *singular values*), and the transpose of an $N \times N$ orthogonal matrix $\mathbf{V}$. The various shapes of these matrices will be made clearer by the following tableau:

$$\begin{pmatrix} & & \\ & \mathbf{A} & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{U} & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} \cdot \begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{pmatrix} \tag{2.6.1}$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ are each orthogonal in the sense that their columns are orthonormal,

$$\sum_{i=1}^{M} U_{ik} U_{in} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.6.2}$$

$$\sum_{j=1}^{N} V_{jk} V_{jn} = \delta_{kn} \qquad \begin{matrix} 1 \le k \le N \\ 1 \le n \le N \end{matrix} \tag{2.6.3}$$

or as a tableau,

$$\begin{pmatrix} & \mathbf{U}^T & \end{pmatrix} \cdot \begin{pmatrix} & \mathbf{U} & \end{pmatrix} = \begin{pmatrix} & \mathbf{V}^T & \end{pmatrix} \cdot \begin{pmatrix} & \mathbf{V} & \end{pmatrix}$$

$$= \begin{pmatrix} & \mathbf{1} & \end{pmatrix} \tag{2.6.4}$$

Since $\mathbf{V}$ is square, it is also row-orthonormal, $\mathbf{V} \cdot \mathbf{V}^T = 1$.

The SVD decomposition can also be carried out when $M < N$. In this case the singular values w_j for $j = M+1, \ldots, N$ are all zero, and the corresponding columns of $\mathbf{U}$ are also zero. Equation (2.6.2) then holds only for $k, n \leq M$.

The decomposition (2.6.1) can always be done, no matter how singular the matrix is, and it is "almost" unique. That is to say, it is unique up to (i) making the same permutation of the columns of $\mathbf{U}$, elements of $\mathbf{W}$, and columns of $\mathbf{V}$ (or rows of $\mathbf{V}^T$), or (ii) forming linear combinations of any columns of $\mathbf{U}$ and $\mathbf{V}$ whose corresponding elements of $\mathbf{W}$ happen to be exactly equal. An important consequence of the permutation freedom is that for the case $M < N$, a numerical algorithm for the decomposition need not return zero w_j's for $j = M+1, \ldots, N$; the $N - M$ zero singular values can be scattered among all positions $j = 1, 2, \ldots, N$.

At the end of this section, we give a routine, `svdcmp`, that performs SVD on an arbitrary matrix $\mathbf{A}$, replacing it by $\mathbf{U}$ (they are the same shape) and returning $\mathbf{W}$ and $\mathbf{V}$ separately. The routine `svdcmp` is based on a routine by Forsythe et al. [1], which is in turn based on the original routine of Golub and Reinsch, found, in various forms, in [2-4] and elsewhere. These references include extensive discussion of the algorithm used. As much as we dislike the use of black-box routines, we are going to ask you to accept this one, since it would take us too far afield to cover its necessary background material here. Suffice it to say that the algorithm is very stable, and that it is very unusual for it ever to misbehave. Most of the concepts that enter the algorithm (Householder reduction to bidiagonal form, diagonalization by QR procedure with shifts) will be discussed further in Chapter 11.

If you are as suspicious of black boxes as we are, you will want to verify yourself that `svdcmp` does what we say it does. That is very easy to do: Generate an arbitrary matrix $\mathbf{A}$, call the routine, and then verify by matrix multiplication that (2.6.1) and (2.6.4) are satisfied. Since these two equations are the only defining requirements for SVD, this procedure is (for the chosen $\mathbf{A}$) a complete end-to-end check.

Now let us find out what SVD is good for.

SVD of a Square Matrix

If the matrix **A** is square, $N \times N$ say, then **U**, **V**, and **W** are all square matrices of the same size. Their inverses are also trivial to compute: **U** and **V** are orthogonal, so their inverses are equal to their transposes; **W** is diagonal, so its inverse is the diagonal matrix whose elements are the reciprocals of the elements w_j. From (2.6.1) it now follows immediately that the inverse of **A** is

$$\mathbf{A}^{-1} = \mathbf{V} \cdot [\text{diag}\,(1/w_j)] \cdot \mathbf{U}^T \tag{2.6.5}$$

The only thing that can go wrong with this construction is for one of the w_j's to be zero, or (numerically) for it to be so small that its value is dominated by roundoff error and therefore unknowable. If more than one of the w_j's have this problem, then the matrix is even more singular. So, first of all, SVD gives you a clear diagnosis of the situation.

Formally, the *condition number* of a matrix is defined as the ratio of the largest (in magnitude) of the w_j's to the smallest of the w_j's. A matrix is singular if its condition number is infinite, and it is *ill-conditioned* if its condition number is too large, that is, if its reciprocal approaches the machine's floating-point precision (for example, less than 10^{-6} for single precision or 10^{-12} for double).

For singular matrices, the concepts of *nullspace* and *range* are important. Consider the familiar set of simultaneous equations

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.6.6}$$

where **A** is a square matrix, **b** and **x** are vectors. Equation (2.6.6) defines **A** as a linear mapping from the vector space **x** to the vector space **b**. If **A** is singular, then there is some subspace of **x**, called the nullspace, that is mapped to zero, $\mathbf{A} \cdot \mathbf{x} = 0$. The dimension of the nullspace (the number of linearly independent vectors **x** that can be found in it) is called the *nullity* of **A**.

Now, there is also some subspace of **b** that can be "reached" by **A**, in the sense that there exists some **x** which is mapped there. This subspace of **b** is called the range of **A**. The dimension of the range is called the *rank* of **A**. If **A** is nonsingular, then its range will be all of the vector space **b**, so its rank is N. If **A** is singular, then the rank will be less than N. In fact, the relevant theorem is "rank plus nullity equals N."

What has this to do with SVD? SVD explicitly constructs orthonormal bases for the nullspace and range of a matrix. Specifically, the columns of **U** whose same-numbered elements w_j are *nonzero* are an orthonormal set of basis vectors that span the range; the columns of **V** whose same-numbered elements w_j are *zero* are an orthonormal basis for the nullspace.

Now let's have another look at solving the set of simultaneous linear equations (2.6.6) in the case that **A** is singular. First, the set of *homogeneous* equations, where $\mathbf{b} = 0$, is solved immediately by SVD: Any column of **V** whose corresponding w_j is zero yields a solution.

When the vector **b** on the right-hand side is not zero, the important question is whether it lies in the range of **A** or not. If it does, then the singular set of equations *does* have a solution **x**; in fact it has more than one solution, since any vector in the nullspace (any column of **V** with a corresponding zero w_j) can be added to **x** in any linear combination.

If we want to single out one particular member of this solution-set of vectors as a representative, we might want to pick the one with the smallest length $|\mathbf{x}|^2$. Here is how to find that vector using SVD: Simply *replace* $1/w_j$ *by zero if* $w_j = 0$. (It is not very often that one gets to set $\infty = 0$!) Then compute (working from right to left)

$$\mathbf{x} = \mathbf{V} \cdot [\text{diag}\,(1/w_j)] \cdot (\mathbf{U}^T \cdot \mathbf{b}) \tag{2.6.7}$$

This will be the solution vector of smallest length; the columns of $\mathbf{V}$ that are in the nullspace complete the specification of the solution set.

Proof: Consider $|\mathbf{x} + \mathbf{x}'|$, where $\mathbf{x}'$ lies in the nullspace. Then, if $\mathbf{W}^{-1}$ denotes the modified inverse of $\mathbf{W}$ with some elements zeroed,

$$\begin{aligned} |\mathbf{x} + \mathbf{x}'| &= \left|\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{x}'\right| \\ &= \left|\mathbf{V} \cdot (\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}')\right| \\ &= \left|\mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{V}^T \cdot \mathbf{x}'\right| \end{aligned} \tag{2.6.8}$$

Here the first equality follows from (2.6.7), the second and third from the orthonormality of $\mathbf{V}$. If you now examine the two terms that make up the sum on the right-hand side, you will see that the first one has nonzero j components only where $w_j \neq 0$, while the second one, since $\mathbf{x}'$ is in the nullspace, has nonzero j components only where $w_j = 0$. Therefore the minimum length obtains for $\mathbf{x}' = 0$, q.e.d.

If $\mathbf{b}$ is not in the range of the singular matrix $\mathbf{A}$, then the set of equations (2.6.6) has no solution. But here is some good news: If $\mathbf{b}$ is not in the range of $\mathbf{A}$, then equation (2.6.7) can still be used to construct a "solution" vector $\mathbf{x}$. This vector $\mathbf{x}$ will not exactly solve $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. But, among all possible vectors $\mathbf{x}$, it will do the closest possible job in the least squares sense. In other words (2.6.7) finds

$$\mathbf{x} \quad \text{which minimizes} \quad r \equiv |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}| \tag{2.6.9}$$

The number r is called the *residual* of the solution.

The proof is similar to (2.6.8): Suppose we modify $\mathbf{x}$ by adding some arbitrary $\mathbf{x}'$. Then $\mathbf{A} \cdot \mathbf{x} - \mathbf{b}$ is modified by adding some $\mathbf{b}' \equiv \mathbf{A} \cdot \mathbf{x}'$. Obviously $\mathbf{b}'$ is in the range of $\mathbf{A}$. We then have

$$\begin{aligned} \left|\mathbf{A} \cdot \mathbf{x} - \mathbf{b} + \mathbf{b}'\right| &= \left|(\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^T) \cdot (\mathbf{V} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T \cdot \mathbf{b}) - \mathbf{b} + \mathbf{b}'\right| \\ &= \left|(\mathbf{U} \cdot \mathbf{W} \cdot \mathbf{W}^{-1} \cdot \mathbf{U}^T - 1) \cdot \mathbf{b} + \mathbf{b}'\right| \\ &= \left|\mathbf{U} \cdot \left[(\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}'\right]\right| \\ &= \left|(\mathbf{W} \cdot \mathbf{W}^{-1} - 1) \cdot \mathbf{U}^T \cdot \mathbf{b} + \mathbf{U}^T \cdot \mathbf{b}'\right| \end{aligned} \tag{2.6.10}$$

Now, $(\mathbf{W} \cdot \mathbf{W}^{-1} - 1)$ is a diagonal matrix which has nonzero j components only for $w_j = 0$, while $\mathbf{U}^T\mathbf{b}'$ has nonzero j components only for $w_j \neq 0$, since $\mathbf{b}'$ lies in the range of $\mathbf{A}$. Therefore the minimum obtains for $\mathbf{b}' = 0$, q.e.d.

Figure 2.6.1 summarizes our discussion of SVD thus far.

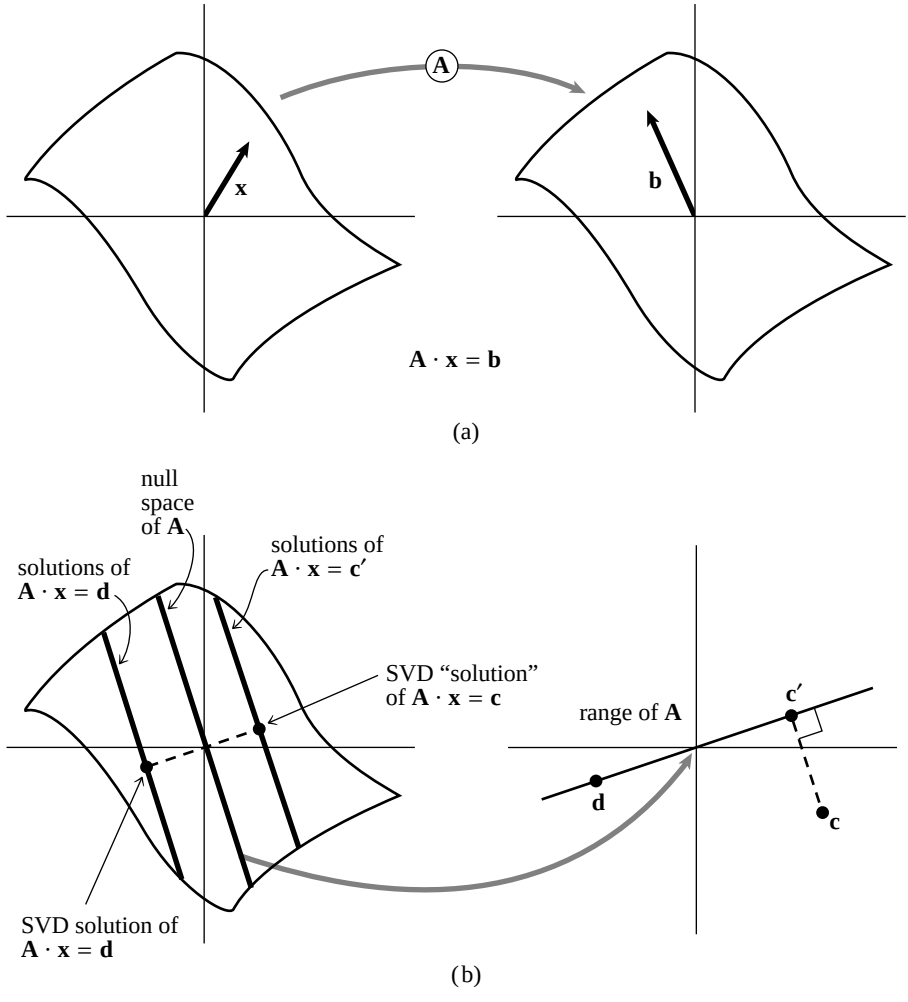


Figure 2.6.1. (a) A nonsingular matrix **A** maps a vector space into one of the same dimension. The vector **x** is mapped into **b**, so that **x** satisfies the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. (b) A singular matrix **A** maps a vector space into one of lower dimensionality, here a plane into a line, called the "range" of **A**. The "nullspace" of **A** is mapped to zero. The solutions of $\mathbf{A} \cdot \mathbf{x} = \mathbf{d}$ consist of any one particular solution plus any vector in the nullspace, here forming a line parallel to the nullspace. Singular value decomposition (SVD) selects the particular solution closest to zero, as shown. The point **c** lies outside of the range of **A**, so $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}$ has no solution. SVD finds the least-squares best compromise solution, namely a solution of $\mathbf{A} \cdot \mathbf{x} = \mathbf{c}'$, as shown.

In the discussion since equation (2.6.6), we have been pretending that a matrix either is singular or else isn't. That is of course true analytically. Numerically, however, the far more common situation is that some of the w_j's are very small but nonzero, so that the matrix is ill-conditioned. In that case, the direct solution methods of LU decomposition or Gaussian elimination may actually give a formal solution to the set of equations (that is, a zero pivot may not be encountered); but the solution vector may have wildly large components whose algebraic cancellation, when multiplying by the matrix **A**, may give a very poor approximation to the right-hand vector **b**. In such cases, the solution vector **x** obtained by *zeroing* the

small w_j's and then using equation (2.6.7) is very often better (in the sense of the residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ being smaller) than *both* the direct-method solution *and* the SVD solution where the small w_j's are left nonzero.

It may seem paradoxical that this can be so, since zeroing a singular value corresponds to throwing away one linear combination of the set of equations that we are trying to solve. The resolution of the paradox is that we are throwing away precisely a combination of equations that is so corrupted by roundoff error as to be at best useless; usually it is worse than useless since it "pulls" the solution vector way off towards infinity along some direction that is almost a nullspace vector. In doing this, it compounds the roundoff problem and makes the residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ larger.

SVD cannot be applied blindly, then. You have to exercise some discretion in deciding at what threshold to zero the small w_j's, and/or you have to have some idea what size of computed residual $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$ is acceptable.

As an example, here is a "backsubstitution" routine `svbksb` for evaluating equation (2.6.7) and obtaining a solution vector **x** from a right-hand side **b**, given that the SVD of a matrix **A** has already been calculated by a call to `svdcmp`. Note that this routine presumes that *you* have already zeroed the small w_j's. It does not do this for you. If you *haven't* zeroed the small w_j's, then this routine is just as ill-conditioned as any direct method, and you are misusing SVD.

```
SUBROUTINE svbksb(u,w,v,m,n,mp,np,b,x)
INTEGER m,mp,n,np,NMAX
REAL b(mp),u(mp,np),v(np,np),w(np),x(np)
PARAMETER (NMAX=500)                    Maximum anticipated value of n.
    Solves A · X = B for a vector X, where A is specified by the arrays u, w, v as returned by
    svdcmp. m and n are the logical dimensions of a, and will be equal for square matrices. mp
    and np are the physical dimensions of a. b(1:m) is the input right-hand side. x(1:n) is
    the output solution vector. No input quantities are destroyed, so the routine may be called
    sequentially with different b's.
INTEGER i,j,jj
REAL s,tmp(NMAX)
do 12 j=1,n                             Calculate U^T B.
    s=0.
    if(w(j).ne.0.)then                  Nonzero result only if w_j is nonzero.
        do 11 i=1,m
            s=s+u(i,j)*b(i)
        enddo 11
        s=s/w(j)                        This is the divide by w_j.
    endif
    tmp(j)=s
enddo 12
do 14 j=1,n                             Matrix multiply by V to get answer.
    s=0.
    do 13 jj=1,n
        s=s+v(j,jj)*tmp(jj)
    enddo 13
    x(j)=s
enddo 14
return
END
```

Note that a typical use of `svdcmp` and `svbksb` superficially resembles the typical use of `ludcmp` and `lubksb`: In both cases, you decompose the left-hand matrix **A** just once, and then can use the decomposition either once or many times with different right-hand sides. The crucial difference is the "editing" of the singular

values before svbksb is called:

```
REAL a(np,np),u(np,np),w(np),v(np,np),b(np),x(np)
...
do 12 i=1,n                          Copy a into u if you don't want it to be destroyed.
    do 11 j=1,n
        u(i,j)=a(i,j)
    enddo 11
enddo 12
call svdcmp(u,n,n,np,np,w,v)         SVD the square matrix a.
wmax=0.                              Will be the maximum singular value obtained.
do 13 j=1,n
    if(w(j).gt.wmax)wmax=w(j)
enddo 13
wmin=wmax*1.0e-6                     This is where we set the threshold for singular values
do 14 j=1,n                              allowed to be nonzero. The constant is typical,
    if(w(j).lt.wmin)w(j)=0.              but not universal. You have to experiment with
enddo 14                                 your own application.
call svbksb(u,w,v,n,n,np,np,b,x)     Now we can backsubstitute.
```

SVD for Fewer Equations than Unknowns

If you have fewer linear equations M than unknowns N, then you are not expecting a unique solution. Usually there will be an $N - M$ dimensional family of solutions. If you want to find this whole solution space, then SVD can readily do the job.

The SVD decomposition will yield $N - M$ zero or negligible w_j's, since $M < N$. There may be additional zero w_j's from any degeneracies in your M equations. Be sure that you find this many small w_j's, and zero them before calling svbksb, which will give you the particular solution vector **x**. As before, the columns of **V** corresponding to zeroed w_j's are the basis vectors whose linear combinations, added to the particular solution, span the solution space.

SVD for More Equations than Unknowns

This situation will occur in Chapter 15, when we wish to find the least-squares solution to an overdetermined set of linear equations. In tableau, the equations to be solved are

$$\begin{pmatrix} & & \\ & \mathbf{A} & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{x} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{b} \\ \\ \end{pmatrix} \tag{2.6.11}$$

The proofs that we gave above for the square case apply without modification to the case of more equations than unknowns. The least-squares solution vector **x** is

given by (2.6.7), which, with nonsquare matrices, looks like this,

$$\begin{pmatrix} \\ \mathbf{x} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{V} \\ \\ \end{pmatrix} \cdot \begin{pmatrix} \\ \mathrm{diag}(1/w_j) \\ \\ \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{U}^T \\ \\ \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{b} \\ \\ \end{pmatrix} \tag{2.6.12}$$

In general, the matrix **W** will not be singular, and no w_j's will need to be set to zero. Occasionally, however, there might be column degeneracies in **A**. In this case you will need to zero some small w_j values after all. The corresponding column in **V** gives the linear combination of **x**'s that is then ill-determined even by the supposedly overdetermined set.

Sometimes, although you do not need to zero any w_j's for *computational* reasons, you may nevertheless want to take note of any that are unusually small: Their corresponding columns in **V** are linear combinations of **x**'s which are insensitive to your data. In fact, you may then wish to zero these w_j's, to reduce the number of free parameters in the fit. These matters are discussed more fully in Chapter 15.

Constructing an Orthonormal Basis

Suppose that you have N vectors in an M-dimensional vector space, with $N \leq M$. Then the N vectors span some subspace of the full vector space. Often you want to construct an orthonormal set of N vectors that span the same subspace. The textbook way to do this is by Gram-Schmidt orthogonalization, starting with one vector and then expanding the subspace one dimension at a time. Numerically, however, because of the build-up of roundoff errors, naive Gram-Schmidt orthogonalization is *terrible.*

The right way to construct an orthonormal basis for a subspace is by SVD: Form an $M \times N$ matrix **A** whose N columns are your vectors. Run the matrix through `svdcmp`. The columns of the matrix **U** (which in fact replaces **A** on output from `svdcmp`) are your desired orthonormal basis vectors.

You might also want to check the output w_j's for zero values. If any occur, then the spanned subspace was not, in fact, N dimensional; the columns of **U** corresponding to zero w_j's should be discarded from the orthonormal basis set.

(QR factorization, discussed in §2.10, also constructs an orthonormal basis, see [5].)

Approximation of Matrices

Note that equation (2.6.1) can be rewritten to express any matrix A_{ij} as a sum of outer products of columns of **U** and rows of $\mathbf{V}^T$, with the "weighting factors" being the singular values w_j,

$$A_{ij} = \sum_{k=1}^{N} w_k \, U_{ik} V_{jk} \tag{2.6.13}$$

If you ever encounter a situation where *most* of the singular values w_j of a matrix **A** are very small, then **A** will be well-approximated by only a few terms in the sum (2.6.13). This means that you have to store only a few columns of **U** and **V** (the same k ones) and you will be able to recover, with good accuracy, the whole matrix.

Note also that it is very efficient to multiply such an approximated matrix by a vector **x**: You just dot **x** with each of the stored columns of **V**, multiply the resulting scalar by the corresponding w_k, and accumulate that multiple of the corresponding column of **U**. If your matrix is approximated by a small number K of singular values, then this computation of $\mathbf{A} \cdot \mathbf{x}$ takes only about $K(M + N)$ multiplications, instead of MN for the full matrix.

SVD Algorithm

Here is the algorithm for constructing the singular value decomposition of any matrix. See §11.2–§11.3, and also [4-5], for discussion relating to the underlying method.

```
      SUBROUTINE svdcmp(a,m,n,mp,np,w,v)
      INTEGER m,mp,n,np,NMAX
      REAL a(mp,np),v(np,np),w(np)
      PARAMETER (NMAX=500)                    Maximum anticipated value of n.
C     USES pythag
```

Given a matrix `a(1:m,1:n)`, with physical dimensions `mp` by `np`, this routine computes its singular value decomposition, $A = U \cdot W \cdot V^T$. The matrix U replaces `a` on output. The diagonal matrix of singular values W is output as a vector `w(1:n)`. The matrix V (not the transpose V^T) is output as `v(1:n,1:n)`.

```
      INTEGER i,its,j,jj,k,l,nm
      REAL anorm,c,f,g,h,s,scale,x,y,z,rv1(NMAX),pythag
      g=0.0                                   Householder reduction to bidiagonal form.
      scale=0.0
      anorm=0.0
      do 25 i=1,n
        l=i+1
        rv1(i)=scale*g
        g=0.0
        s=0.0
        scale=0.0
        if(i.le.m)then
          do 11 k=i,m
            scale=scale+abs(a(k,i))
          enddo 11
          if(scale.ne.0.0)then
            do 12 k=i,m
              a(k,i)=a(k,i)/scale
              s=s+a(k,i)*a(k,i)
            enddo 12
            f=a(i,i)
            g=-sign(sqrt(s),f)
            h=f*g-s
            a(i,i)=f-g
            do 15 j=l,n
              s=0.0
              do 13 k=i,m
                s=s+a(k,i)*a(k,j)
              enddo 13
              f=s/h
              do 14 k=i,m
                a(k,j)=a(k,j)+f*a(k,i)
              enddo 14
```

```
            enddo 15
            do 16 k=i,m
                a(k,i)=scale*a(k,i)
            enddo 16
        endif
    endif
    w(i)=scale *g
    g=0.0
    s=0.0
    scale=0.0
    if((i.le.m).and.(i.ne.n))then
        do 17 k=l,n
            scale=scale+abs(a(i,k))
        enddo 17
        if(scale.ne.0.0)then
            do 18 k=l,n
                a(i,k)=a(i,k)/scale
                s=s+a(i,k)*a(i,k)
            enddo 18
            f=a(i,l)
            g=-sign(sqrt(s),f)
            h=f*g-s
            a(i,l)=f-g
            do 19 k=l,n
                rv1(k)=a(i,k)/h
            enddo 19
            do 23 j=l,m
                s=0.0
                do 21 k=l,n
                    s=s+a(j,k)*a(i,k)
                enddo 21
                do 22 k=l,n
                    a(j,k)=a(j,k)+s*rv1(k)
                enddo 22
            enddo 23
            do 24 k=l,n
                a(i,k)=scale*a(i,k)
            enddo 24
        endif
    endif
    anorm=max(anorm,(abs(w(i))+abs(rv1(i))))
enddo 25
do 32 i=n,1,-1                          Accumulation of right-hand transformations.
    if(i.lt.n)then
        if(g.ne.0.0)then
            do 26 j=l,n                 Double division to avoid possible underflow.
                v(j,i)=(a(i,j)/a(i,l))/g
            enddo 26
            do 29 j=l,n
                s=0.0
                do 27 k=l,n
                    s=s+a(i,k)*v(k,j)
                enddo 27
                do 28 k=l,n
                    v(k,j)=v(k,j)+s*v(k,i)
                enddo 28
            enddo 29
        endif
        do 31 j=l,n
            v(i,j)=0.0
            v(j,i)=0.0
        enddo 31
    endif
    v(i,i)=1.0
```

```
            g=rv1(i)
            l=i
        enddo 32
        do 39 i=min(m,n),1,-1                    Accumulation of left-hand transformations.
            l=i+1
            g=w(i)
            do 33 j=l,n
                a(i,j)=0.0
            enddo 33
            if(g.ne.0.0)then
                g=1.0/g
                do 36 j=l,n
                    s=0.0
                    do 34 k=l,m
                        s=s+a(k,i)*a(k,j)
                    enddo 34
                    f=(s/a(i,i))*g
                    do 35 k=i,m
                        a(k,j)=a(k,j)+f*a(k,i)
                    enddo 35
                enddo 36
                do 37 j=i,m
                    a(j,i)=a(j,i)*g
                enddo 37
            else
                do 38 j= i,m
                    a(j,i)=0.0
                enddo 38
            endif
            a(i,i)=a(i,i)+1.0
        enddo 39
        do 49 k=n,1,-1                           Diagonalization of the bidiagonal form: Loop over
            do 48 its=1,30                           singular values, and over allowed iterations.
                do 41 l=k,1,-1                   Test for splitting.
                    nm=l-1                       Note that rv1(1) is always zero.
                    if((abs(rv1(l))+anorm).eq.anorm) goto 2
                    if((abs(w(nm))+anorm).eq.anorm) goto 1
                enddo 41
1               c=0.0                            Cancellation of rv1(l), if l > 1.
                s=1.0
                do 43 i=l,k
                    f=s*rv1(i)
                    rv1(i)=c*rv1(i)
                    if((abs(f)+anorm).eq.anorm) goto 2
                    g=w(i)
                    h=pythag(f,g)
                    w(i)=h
                    h=1.0/h
                    c= (g*h)
                    s=-(f*h)
                    do 42 j=1,m
                        y=a(j,nm)
                        z=a(j,i)
                        a(j,nm)=(y*c)+(z*s)
                        a(j,i)=-(y*s)+(z*c)
                    enddo 42
                enddo 43
2               z=w(k)
                if(l.eq.k)then                   Convergence.
                    if(z.lt.0.0)then             Singular value is made nonnegative.
                        w(k)=-z
                        do 44 j=1,n
                            v(j,k)=-v(j,k)
                        enddo 44
```

```
            endif
            goto 3
        endif
        if(its.eq.30) pause 'no convergence in svdcmp'
        x=w(l)                          Shift from bottom 2-by-2 minor.
        nm=k-1
        y=w(nm)
        g=rv1(nm)
        h=rv1(k)
        f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y)
        g=pythag(f,1.0)
        f=((x-z)*(x+z)+h*((y/(f+sign(g,f)))-h))/x
        c=1.0                           Next QR transformation:
        s=1.0
        do 47 j=l,nm
            i=j+1
            g=rv1(i)
            y=w(i)
            h=s*g
            g=c*g
            z=pythag(f,h)
            rv1(j)=z
            c=f/z
            s=h/z
            f= (x*c)+(g*s)
            g=-(x*s)+(g*c)
            h=y*s
            y=y*c
            do 45 jj=1,n
                x=v(jj,j)
                z=v(jj,i)
                v(jj,j)= (x*c)+(z*s)
                v(jj,i)=-(x*s)+(z*c)
            enddo 45
            z=pythag(f,h)
            w(j)=z                      Rotation can be arbitrary if z = 0.
            if(z.ne.0.0)then
                z=1.0/z
                c=f*z
                s=h*z
            endif
            f= (c*g)+(s*y)
            x=-(s*g)+(c*y)
            do 46 jj=1,m
                y=a(jj,j)
                z=a(jj,i)
                a(jj,j)= (y*c)+(z*s)
                a(jj,i)=-(y*s)+(z*c)
            enddo 46
        enddo 47
        rv1(l)=0.0
        rv1(k)=f
        w(k)=x
    enddo 48
3   continue
enddo 49
return
END


FUNCTION pythag(a,b)
REAL a,b,pythag
```

Computes $(a^2 + b^2)^{1/2}$ without destructive underflow or overflow.

```
REAL absa,absb
absa=abs(a)
absb=abs(b)
if(absa.gt.absb)then
    pythag=absa*sqrt(1.+(absb/absa)**2)
else
    if(absb.eq.0.)then
        pythag=0.
    else
        pythag=absb*sqrt(1.+(absa/absb)**2)
    endif
endif
return
END
```

(Double precision versions of `svdcmp`, `svbksb`, and `pythag`, named `dsvdcmp`, `dsvbksb`, and `dpythag`, are used by the routine `ratlsq` in §5.13. You can easily make the conversions, or else get the converted routines from the *Numerical Recipes* diskette.)

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.3 and Chapter 12.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 18.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I.10 by G.H. Golub and C. Reinsch. [2]

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.), Chapter 11. [3]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.7. [4]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.2.6. [5]

2.7 Sparse Linear Systems

A system of linear equations is called *sparse* if only a relatively small number of its matrix elements a_{ij} are nonzero. It is wasteful to use general methods of linear algebra on such problems, because most of the $O(N^3)$ arithmetic operations devoted to solving the set of equations or inverting the matrix involve zero operands. Furthermore, you might wish to work problems so large as to tax your available memory space, and it is wasteful to reserve storage for unfruitful zero elements. Note that there are two distinct (and not always compatible) goals for any sparse matrix method: saving time and/or saving space.

We have already considered one archetypal sparse form in §2.4, the band diagonal matrix. In the tridiagonal case, e.g., we saw that it was possible to save

```
REAL absa,absb
absa=abs(a)
absb=abs(b)
if(absa.gt.absb)then
    pythag=absa*sqrt(1.+(absb/absa)**2)
else
    if(absb.eq.0.)then
        pythag=0.
    else
        pythag=absb*sqrt(1.+(absa/absb)**2)
    endif
endif
return
END
```

(Double precision versions of `svdcmp`, `svbksb`, and `pythag`, named `dsvdcmp`, `dsvbksb`, and `dpythag`, are used by the routine `ratlsq` in §5.13. You can easily make the conversions, or else get the converted routines from the *Numerical Recipes* diskette.)

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.3 and Chapter 12.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 18.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I.10 by G.H. Golub and C. Reinsch. [2]

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.), Chapter 11. [3]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.7. [4]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.2.6. [5]

2.7 Sparse Linear Systems

A system of linear equations is called *sparse* if only a relatively small number of its matrix elements a_{ij} are nonzero. It is wasteful to use general methods of linear algebra on such problems, because most of the $O(N^3)$ arithmetic operations devoted to solving the set of equations or inverting the matrix involve zero operands. Furthermore, you might wish to work problems so large as to tax your available memory space, and it is wasteful to reserve storage for unfruitful zero elements. Note that there are two distinct (and not always compatible) goals for any sparse matrix method: saving time and/or saving space.

We have already considered one archetypal sparse form in §2.4, the band diagonal matrix. In the tridiagonal case, e.g., we saw that it was possible to save

both time (order N instead of N^3) and space (order N instead of N^2). The method of solution was not different in principle from the general method of LU decomposition; it was just applied cleverly, and with due attention to the bookkeeping of zero elements. Many practical schemes for dealing with sparse problems have this same character. They are fundamentally decomposition schemes, or else elimination schemes akin to Gauss-Jordan, but carefully optimized so as to minimize the number of so-called *fill-ins*, initially zero elements which must become nonzero during the solution process, and for which storage must be reserved.

Direct methods for solving sparse equations, then, depend crucially on the precise pattern of sparsity of the matrix. Patterns that occur frequently, or that are useful as way-stations in the reduction of more general forms, already have special names and special methods of solution. We do not have space here for any detailed review of these. References listed at the end of this section will furnish you with an "in" to the specialized literature, and the following list of buzz words (and Figure 2.7.1) will at least let you hold your own at cocktail parties:

- tridiagonal
- band diagonal (or banded) with bandwidth M
- band triangular
- block diagonal
- block tridiagonal
- block triangular
- cyclic banded
- singly (or doubly) bordered block diagonal
- singly (or doubly) bordered block triangular
- singly (or doubly) bordered band diagonal
- singly (or doubly) bordered band triangular
- other (!)

You should also be aware of some of the special sparse forms that occur in the solution of partial differential equations in two or more dimensions. See Chapter 19.

If your particular pattern of sparsity is not a simple one, then you may wish to try an *analyze/factorize/operate* package, which automates the procedure of figuring out how fill-ins are to be minimized. The *analyze* stage is done once only for each pattern of sparsity. The *factorize* stage is done once for each particular matrix that fits the pattern. The *operate* stage is performed once for each right-hand side to be used with the particular matrix. Consult [2,3] for references on this. The NAG library [4] has an analyze/factorize/operate capability. A substantial collection of routines for sparse matrix calculation is also available from IMSL [5] as the *Yale Sparse Matrix Package* [6].

You should be aware that the special order of interchanges and eliminations, prescribed by a sparse matrix method so as to minimize fill-ins and arithmetic operations, generally acts to decrease the method's numerical stability as compared to, e.g., regular LU decomposition with pivoting. Scaling your problem so as to make its nonzero matrix elements have comparable magnitudes (if you can do it) will sometimes ameliorate this problem.

In the remainder of this section, we present some concepts which are applicable to some general classes of sparse matrices, and which do not necessarily depend on details of the pattern of sparsity.

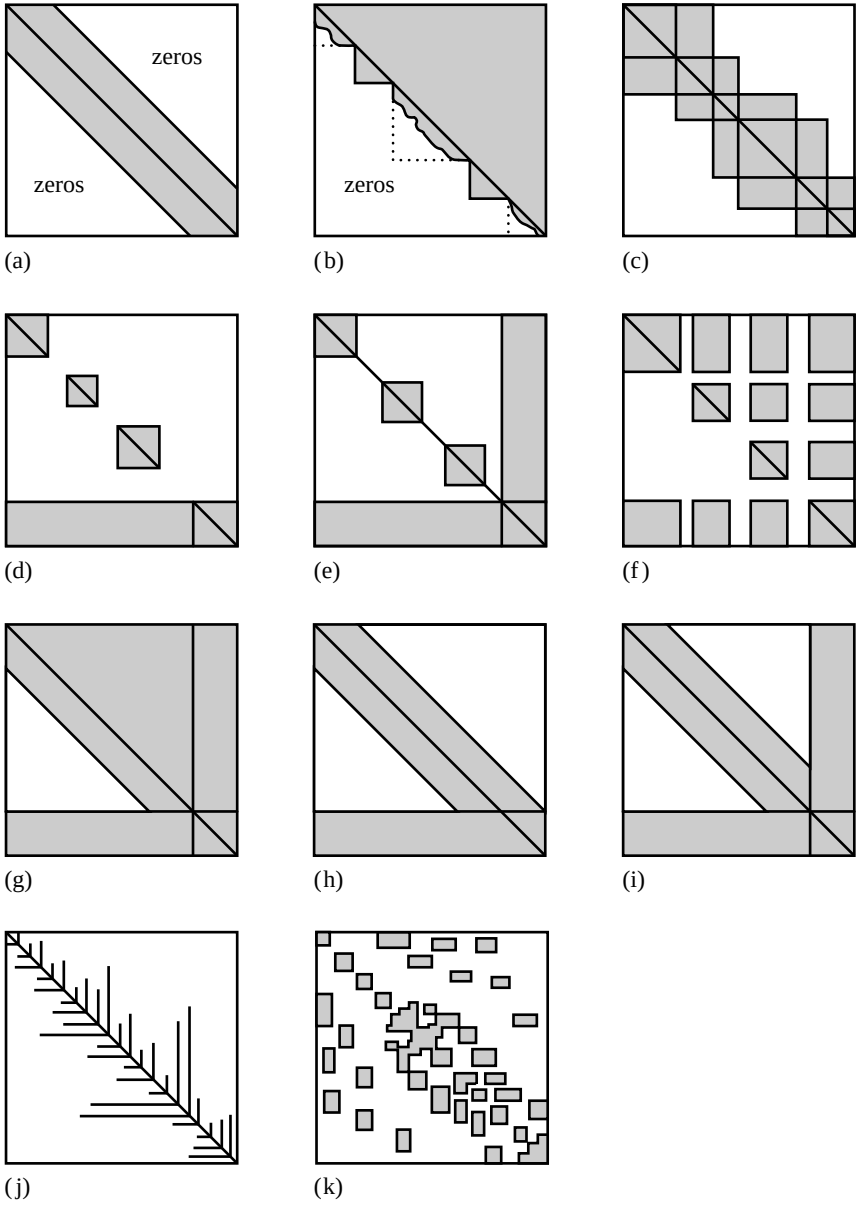


Figure 2.7.1. Some standard forms for sparse matrices. (a) Band diagonal; (b) block triangular; (c) block tridiagonal; (d) singly bordered block diagonal; (e) doubly bordered block diagonal; (f) singly bordered block triangular; (g) bordered band-triangular; (h) and (i) singly and doubly bordered band diagonal; (j) and (k) other! (after Tewarson) [1].

Sherman-Morrison Formula

Suppose that you have already obtained, by herculean effort, the inverse matrix $\mathbf{A}^{-1}$ of a square matrix $\mathbf{A}$. Now you want to make a "small" change in $\mathbf{A}$, for example change one element a_{ij}, or a few elements, or one row, or one column. Is there any way of calculating the corresponding change in $\mathbf{A}^{-1}$ without repeating

your difficult labors? Yes, if your change is of the form

$$\mathbf{A} \rightarrow (\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \tag{2.7.1}$$

for some vectors $\mathbf{u}$ and $\mathbf{v}$. If $\mathbf{u}$ is a unit vector $\mathbf{e}_i$, then (2.7.1) adds the components of $\mathbf{v}$ to the ith row. (Recall that $\mathbf{u} \otimes \mathbf{v}$ is a matrix whose i, jth element is the product of the ith component of $\mathbf{u}$ and the jth component of $\mathbf{v}$.) If $\mathbf{v}$ is a unit vector $\mathbf{e}_j$, then (2.7.1) adds the components of $\mathbf{u}$ to the jth column. If both $\mathbf{u}$ and $\mathbf{v}$ are proportional to unit vectors $\mathbf{e}_i$ and $\mathbf{e}_j$ respectively, then a term is added only to the element a_{ij}.

The *Sherman-Morrison* formula gives the inverse $(\mathbf{A}+\mathbf{u}\otimes\mathbf{v})^{-1}$, and is derived briefly as follows:

$$\begin{aligned}
(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1} &= (\mathbf{1} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v})^{-1} \cdot \mathbf{A}^{-1} \\
&= (\mathbf{1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} - \ldots) \cdot \mathbf{A}^{-1} \\
&= \mathbf{A}^{-1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \left(1 - \lambda + \lambda^2 - \ldots\right) \\
&= \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1} \cdot \mathbf{u}) \otimes (\mathbf{v} \cdot \mathbf{A}^{-1})}{1 + \lambda}
\end{aligned} \tag{2.7.2}$$

where

$$\lambda \equiv \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \tag{2.7.3}$$

The second line of (2.7.2) is a formal power series expansion. In the third line, the associativity of outer and inner products is used to factor out the scalars λ.

The use of (2.7.2) is this: Given $\mathbf{A}^{-1}$ and the vectors $\mathbf{u}$ and $\mathbf{v}$, we need only perform two matrix multiplications and a vector dot product,

$$\mathbf{z} \equiv \mathbf{A}^{-1} \cdot \mathbf{u} \qquad \mathbf{w} \equiv (\mathbf{A}^{-1})^T \cdot \mathbf{v} \qquad \lambda = \mathbf{v} \cdot \mathbf{z} \tag{2.7.4}$$

to get the desired change in the inverse

$$\mathbf{A}^{-1} \quad \rightarrow \quad \mathbf{A}^{-1} - \frac{\mathbf{z} \otimes \mathbf{w}}{1 + \lambda} \tag{2.7.5}$$

The whole procedure requires only $3N^2$ multiplies and a like number of adds (an even smaller number if $\mathbf{u}$ or $\mathbf{v}$ is a unit vector).

The Sherman-Morrison formula can be directly applied to a class of sparse problems. If you already have a fast way of calculating the inverse of $\mathbf{A}$ (e.g., a tridiagonal matrix, or some other standard sparse form), then (2.7.4)–(2.7.5) allow you to build up to your related but more complicated form, adding for example a row or column at a time. Notice that you can apply the Sherman-Morrison formula more than once successively, using at each stage the most recent update of $\mathbf{A}^{-1}$ (equation 2.7.5). Of course, if you have to modify *every* row, then you are back to an N^3 method. The constant in front of the N^3 is only a few times worse than the better direct methods, but you have deprived yourself of the stabilizing advantages of pivoting — so be careful.

For some other sparse problems, the Sherman-Morrison formula cannot be directly applied for the simple reason that storage of the whole inverse matrix $\mathbf{A}^{-1}$

is not feasible. If you want to add only a single correction of the form $\mathbf{u} \otimes \mathbf{v}$, and solve the linear system

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.6}$$

then you proceed as follows. Using the fast method that is presumed available for the matrix $\mathbf{A}$, solve the two auxiliary problems

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \qquad \mathbf{A} \cdot \mathbf{z} = \mathbf{u} \tag{2.7.7}$$

for the vectors $\mathbf{y}$ and $\mathbf{z}$. In terms of these,

$$\mathbf{x} = \mathbf{y} - \left[\frac{\mathbf{v} \cdot \mathbf{y}}{1 + (\mathbf{v} \cdot \mathbf{z})} \right] \mathbf{z} \tag{2.7.8}$$

as we see by multiplying (2.7.2) on the right by $\mathbf{b}$.

Cyclic Tridiagonal Systems

So-called *cyclic tridiagonal systems* occur quite frequently, and are a good example of how to use the Sherman-Morrison formula in the manner just described. The equations have the form

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \beta \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ \alpha & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.7.9}$$

This is a tridiagonal system, except for the matrix elements α and β in the corners. Forms like this are typically generated by finite-differencing differential equations with periodic boundary conditions (§19.4).

We use the Sherman-Morrison formula, treating the system as tridiagonal plus a correction. In the notation of equation (2.7.6), define vectors $\mathbf{u}$ and $\mathbf{v}$ to be

$$\mathbf{u} = \begin{bmatrix} \gamma \\ 0 \\ \vdots \\ 0 \\ \alpha \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ \beta/\gamma \end{bmatrix} \tag{2.7.10}$$

Here γ is arbitrary for the moment. Then the matrix $\mathbf{A}$ is the tridiagonal part of the matrix in (2.7.9), with two terms modified:

$$b_1' = b_1 - \gamma, \qquad b_N' = b_N - \alpha\beta/\gamma \tag{2.7.11}$$

We now solve equations (2.7.7) with the standard tridiagonal algorithm, and then get the solution from equation (2.7.8).

The routine `cyclic` below implements this algorithm. We choose the arbitrary parameter $\gamma = -b_1$ to avoid loss of precision by subtraction in the first of equations (2.7.11). In the unlikely event that this causes loss of precision in the second of these equations, you can make a different choice.

```
      SUBROUTINE cyclic(a,b,c,alpha,beta,r,x,n)
      INTEGER n,NMAX
      REAL alpha,beta,a(n),b(n),c(n),r(n),x(n)
      PARAMETER (NMAX=500)
C     USES tridag
        Solves for a vector x(1:n) the "cyclic" set of linear equations given by equation (2.7.9).
        a, b, c, and r are input vectors, while alpha and beta are the corner entries in the matrix.
        The input is not modified.
      INTEGER i
      REAL fact,gamma,bb(NMAX),u(NMAX),z(NMAX)
      if(n.le.2)pause 'n too small in cyclic'
      if(n.gt.NMAX)pause 'NMAX too small in cyclic'
      gamma=-b(1)                       Avoid subtraction error in forming bb(1).
      bb(1)=b(1)-gamma                  Set up the diagonal of the modified tridiagonal system.
      bb(n)=b(n)-alpha*beta/gamma
      do 11 i=2,n-1
        bb(i)=b(i)
      enddo 11
      call tridag(a,bb,c,r,x,n)         Solve A · x = r.
      u(1)=gamma                        Set up the vector u.
      u(n)=alpha
      do 12 i=2,n-1
        u(i)=0.
      enddo 12
      call tridag(a,bb,c,u,z,n)         Solve A · z = u.
      fact=(x(1)+beta*x(n)/gamma)/(1.+z(1)+beta*z(n)/gamma)      Form v · x/(1 + v · z).
      do 13 i=1,n                       Now get the solution vector x.
        x(i)=x(i)-fact*z(i)
      enddo 13
      return
      END
```

Woodbury Formula

If you want to add more than a single correction term, then you cannot use (2.7.8) repeatedly, since without storing a new $\mathbf{A}^{-1}$ you will not be able to solve the auxiliary problems (2.7.7) efficiently after the first step. Instead, you need the *Woodbury formula*, which is the block-matrix version of the Sherman-Morrison formula,

$$\begin{aligned}(\mathbf{A}+\mathbf{U}\cdot\mathbf{V}^T)^{-1} \\ = \mathbf{A}^{-1} \; - \; \left[\mathbf{A}^{-1}\cdot\mathbf{U}\cdot(\mathbf{1}+\mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U})^{-1}\cdot\mathbf{V}^T\cdot\mathbf{A}^{-1}\right]\end{aligned} \tag{2.7.12}$$

Here A is, as usual, an $N \times N$ matrix, while $\mathbf{U}$ and $\mathbf{V}$ are $N \times P$ matrices with $P < N$ and usually $P \ll N$. The inner piece of the correction term may become clearer if written as the tableau,

$$\begin{bmatrix} \\ \\ \\ \mathbf{U} \\ \\ \\ \\ \end{bmatrix} \cdot \left[\mathbf{1}+\mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U}\right]^{-1} \cdot \begin{bmatrix} \\ \mathbf{V}^T \\ \\ \end{bmatrix} \tag{2.7.13}$$

where you can see that the matrix whose inverse is needed is only $P \times P$ rather than $N \times N$.

The relation between the Woodbury formula and successive applications of the Sherman-Morrison formula is now clarified by noting that, if $\mathbf{U}$ is the matrix formed by columns out of the P vectors $\mathbf{u}_1, \ldots, \mathbf{u}_P$, and $\mathbf{V}$ is the matrix formed by columns out of the P vectors $\mathbf{v}_1, \ldots, \mathbf{v}_P$,

$$\mathbf{U} \equiv \begin{bmatrix} \\ \mathbf{u}_1 \\ \\ \end{bmatrix} \cdots \begin{bmatrix} \\ \mathbf{u}_P \\ \\ \end{bmatrix} \qquad \mathbf{V} \equiv \begin{bmatrix} \\ \mathbf{v}_1 \\ \\ \end{bmatrix} \cdots \begin{bmatrix} \\ \mathbf{v}_P \\ \\ \end{bmatrix} \tag{2.7.14}$$

then two ways of expressing the same correction to $\mathbf{A}$ are

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k \right) = (\mathbf{A} + \mathbf{U} \cdot \mathbf{V}^T) \tag{2.7.15}$$

(Note that the subscripts on $\mathbf{u}$ and $\mathbf{v}$ do *not* denote components, but rather distinguish the different column vectors.)

Equation (2.7.15) reveals that, if you have $\mathbf{A}^{-1}$ in storage, then you can either make the P corrections in one fell swoop by using (2.7.12), inverting a $P \times P$ matrix, or else make them by applying (2.7.5) P successive times.

If you don't have storage for $\mathbf{A}^{-1}$, then you *must* use (2.7.12) in the following way: To solve the linear equation

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k \right) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.16}$$

first solve the P auxiliary problems

$$\begin{aligned} \mathbf{A} \cdot \mathbf{z}_1 &= \mathbf{u}_1 \\ \mathbf{A} \cdot \mathbf{z}_2 &= \mathbf{u}_2 \\ &\cdots \\ \mathbf{A} \cdot \mathbf{z}_P &= \mathbf{u}_P \end{aligned} \tag{2.7.17}$$

and construct the matrix $\mathbf{Z}$ by columns from the $\mathbf{z}$'s obtained,

$$\mathbf{Z} \equiv \begin{bmatrix} \\ \mathbf{z}_1 \\ \\ \end{bmatrix} \cdots \begin{bmatrix} \\ \mathbf{z}_P \\ \\ \end{bmatrix} \tag{2.7.18}$$

Next, do the $P \times P$ matrix inversion

$$\mathbf{H} \equiv (\mathbf{1} + \mathbf{V}^T \cdot \mathbf{Z})^{-1} \tag{2.7.19}$$

Finally, solve the one further auxiliary problem

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \tag{2.7.20}$$

In terms of these quantities, the solution is given by

$$\mathbf{x} = \mathbf{y} - \mathbf{Z} \cdot \Big[\mathbf{H} \cdot (\mathbf{V}^T \cdot \mathbf{y}) \Big] \tag{2.7.21}$$

Inversion by Partitioning

Once in a while, you will encounter a matrix (not even necessarily sparse) that can be inverted efficiently by partitioning. Suppose that the $N \times N$ matrix $\mathbf{A}$ is partitioned into

$$\mathbf{A} = \begin{bmatrix} \mathbf{P} & \mathbf{Q} \\ \mathbf{R} & \mathbf{S} \end{bmatrix} \tag{2.7.22}$$

where $\mathbf{P}$ and $\mathbf{S}$ are square matrices of size $p \times p$ and $s \times s$ respectively ($p + s = N$). The matrices $\mathbf{Q}$ and $\mathbf{R}$ are not necessarily square, and have sizes $p \times s$ and $s \times p$, respectively.

If the inverse of $\mathbf{A}$ is partitioned in the same manner,

$$\mathbf{A}^{-1} = \begin{bmatrix} \widetilde{\mathbf{P}} & \widetilde{\mathbf{Q}} \\ \widetilde{\mathbf{R}} & \widetilde{\mathbf{S}} \end{bmatrix} \tag{2.7.23}$$

then $\widetilde{\mathbf{P}}$, $\widetilde{\mathbf{Q}}$, $\widetilde{\mathbf{R}}$, $\widetilde{\mathbf{S}}$, which have the same sizes as $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{S}$, respectively, can be found by either the formulas

$$\begin{aligned} \widetilde{\mathbf{P}} &= (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \widetilde{\mathbf{Q}} &= -(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1}) \\ \widetilde{\mathbf{R}} &= -(\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \widetilde{\mathbf{S}} &= \mathbf{S}^{-1} + (\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1}) \end{aligned} \tag{2.7.24}$$

or else by the equivalent formulas

$$\begin{aligned} \widetilde{\mathbf{P}} &= \mathbf{P}^{-1} + (\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \widetilde{\mathbf{Q}} &= -(\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \\ \widetilde{\mathbf{R}} &= -(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \widetilde{\mathbf{S}} &= (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \end{aligned} \tag{2.7.25}$$

The parentheses in equations (2.7.24) and (2.7.25) highlight repeated factors that you may wish to compute only once. (Of course, by associativity, you can instead do the matrix multiplications in any order you like.) The choice between using equation (2.7.24) and (2.7.25) depends on whether you want $\widetilde{\mathbf{P}}$ or $\widetilde{\mathbf{S}}$ to have the simpler formula; or on whether the repeated expression $(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1}$ is easier to calculate than the expression $(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1}$; or on the relative sizes of $\mathbf{P}$ and $\mathbf{S}$; or on whether $\mathbf{P}^{-1}$ or $\mathbf{S}^{-1}$ is already known.

Another sometimes useful formula is for the determinant of the partitioned matrix,

$$\det \mathbf{A} = \det \mathbf{P} \det(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q}) = \det \mathbf{S} \det(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R}) \tag{2.7.26}$$

Indexed Storage of Sparse Matrices

We have already seen (§2.4) that tri- or band-diagonal matrices can be stored in a compact format that allocates storage only to elements which can be nonzero, plus perhaps a few wasted locations to make the bookkeeping easier. What about more general sparse matrices? When a sparse matrix of logical size $N \times N$ contains only a few times N nonzero elements (a typical case), it is surely inefficient — and often physically impossible — to allocate storage for all N^2 elements. Even if one did allocate such storage, it would be inefficient or prohibitive in machine time to loop over all of it in search of nonzero elements.

Obviously some kind of indexed storage scheme is required, one that stores only nonzero matrix elements, along with sufficient auxiliary information to determine where an element logically belongs and how the various elements can be looped over in common matrix operations. Unfortunately, there is no one standard scheme in general use. Knuth [7] describes one method. The Yale Sparse Matrix Package [6] and ITPACK [8] describe several other methods. For most applications, we favor the storage scheme used by PCGPACK [9], which is almost the same as that described by Bentley [10], and also similar to one of the Yale Sparse Matrix Package methods. The advantage of this scheme, which can be called *row-indexed sparse storage mode*, is that it requires storage of only about two times the number of nonzero matrix elements. (Other methods can require as much as three or five times.) For simplicity, we will treat only the case of square matrices, which occurs most frequently in practice.

To represent a matrix $\mathbf{A}$ of logical size $N \times N$, the row-indexed scheme sets up two one-dimensional arrays, call them `sa` and `ija`. The first of these stores matrix element values in single or double precision as desired; the second stores integer values. The storage rules are:

- The first N locations of `sa` store $\mathbf{A}$'s diagonal matrix elements, in order. (Note that diagonal elements are stored even if they are zero; this is at most a slight storage inefficiency, since diagonal elements are nonzero in most realistic applications.)
- Each of the first N locations of `ija` stores the index of the array `sa` that contains the first *off-diagonal* element of the corresponding row of the matrix. (If there are no off-diagonal elements for that row, it is one greater than the index in `sa` of the most recently stored element of a previous row.)
- Location 1 of `ija` is always equal to $N + 2$. (It can be read to determine N.)
- Location $N + 1$ of `ija` is one greater than the index in `sa` of the last off-diagonal element of the last row. (It can be read to determine the number of nonzero elements in the matrix, or the logical length of the arrays `sa` and `ija`.) Location $N + 1$ of `sa` is not used and can be set arbitrarily.
- Entries in `sa` at locations $\geq N + 2$ contain $\mathbf{A}$'s off-diagonal values, ordered by rows and, within each row, ordered by columns.
- Entries in `ija` at locations $\geq N+2$ contain the column number of the corresponding element in `sa`.

While these rules seem arbitrary at first sight, they result in a rather elegant storage scheme. As an example, consider the matrix

$$\begin{bmatrix} 3. & 0. & 1. & 0. & 0. \\ 0. & 4. & 0. & 0. & 0. \\ 0. & 7. & 5. & 9. & 0. \\ 0. & 0. & 0. & 0. & 2. \\ 0. & 0. & 0. & 6. & 5. \end{bmatrix} \tag{2.7.27}$$

In row-indexed compact storage, matrix (2.7.27) is represented by the two arrays of length 11, as follows

index `k`	1	2	3	4	5	6	7	8	9	10	11
`ija(k)`	7	8	8	10	11	12	3	2	4	5	4
`sa(k)`	3.	4.	5.	0.	5.	x	1.	7.	9.	2.	6.

(2.7.28)

Here x is an arbitrary value. Notice that, according to the storage rules, the value of N (namely 5) is `ija(1)-2`, and the length of each array is `ija(ija(1)-1)-1`, namely 11.

The diagonal element in row i is sa(i), and the off-diagonal elements in that row are in sa(k) where k loops from ija(i) to ija(i+1)-1, if the upper limit is greater or equal to the lower one (as in FORTRAN do loops).

Here is a routine, sprsin, that converts a matrix from full storage mode into row-indexed sparse storage mode, throwing away any elements that are less than a specified threshold. Of course, the principal use of sparse storage mode is for matrices whose full storage mode won't fit into your machine at all; then you have to generate them directly into sparse format. Nevertheless sprsin is useful as a precise algorithmic definition of the storage scheme, for subscale testing of large problems, and for the case where execution time, rather than storage, furnishes the impetus to sparse storage.

```
SUBROUTINE sprsin(a,n,np,thresh,nmax,sa,ija)
INTEGER n,nmax,np,ija(nmax)
REAL thresh,a(np,np),sa(nmax)
    Converts a square matrix a(1:n,1:n) with physical dimension np into row-indexed sparse
    storage mode. Only elements of a with magnitude ≥thresh are retained. Output is in
    two linear arrays with physical dimension nmax (an input parameter): sa(1:) contains
    array values, indexed by ija(1:). The logical sizes of sa and ija on output are both
    ija(ija(1)-1)-1 (see text).
INTEGER i,j,k
do 11 j=1,n                             Store diagonal elements.
    sa(j)=a(j,j)
enddo 11
ija(1)=n+2                              Index to 1st row off-diagonal element, if any.
k=n+1
do 13 i=1,n                             Loop over rows.
    do 12 j=1,n                         Loop over columns.
        if(abs(a(i,j)).ge.thresh)then
            if(i.ne.j)then              Store off-diagonal elements and their columns.
                k=k+1
                if(k.gt.nmax)pause 'nmax too small in sprsin'
                sa(k)=a(i,j)
                ija(k)=j
            endif
        endif
    enddo 12
    ija(i+1)=k+1                        As each row is completed, store index to next.
enddo 13
return
END
```

The single most important use of a matrix in row-indexed sparse storage mode is to multiply a vector to its right. In fact, the storage mode is optimized for just this purpose. The following routine is thus very simple.

```
SUBROUTINE sprsax(sa,ija,x,b,n)
INTEGER n,ija(*)
REAL b(n),sa(*),x(n)
    Multiply a matrix in row-index sparse storage arrays sa and ija by a vector x(1:n), giving
    a vector b(1:n).
INTEGER i,k
if (ija(1).ne.n+2) pause 'mismatched vector and matrix in sprsax'
do 12 i=1,n
    b(i)=sa(i)*x(i)                     Start with diagonal term.
    do 11 k=ija(i),ija(i+1)-1           Loop over off-diagonal terms.
        b(i)=b(i)+sa(k)*x(ija(k))
    enddo 11
enddo 12
return
END
```

It is also simple to multiply the *transpose* of a matrix by a vector to its right. (We will use this operation later in this section.) Note that the transpose matrix is not actually constructed.

```
      SUBROUTINE sprstx(sa,ija,x,b,n)
      INTEGER n,ija(*)
      REAL b(n),sa(*),x(n)
        Multiply the transpose of a matrix in row-index sparse storage arrays sa and ija by a
        vector x(1:n), giving a vector b(1:n).
      INTEGER i,j,k
      if (ija(1).ne.n+2) pause 'mismatched vector and matrix in sprstx'
      do 11 i=1,n                    Start with diagonal terms.
        b(i)=sa(i)*x(i)
      enddo 11
      do 13 i=1,n                    Loop over off-diagonal terms.
        do 12 k=ija(i),ija(i+1)-1
          j=ija(k)
          b(j)=b(j)+sa(k)*x(i)
        enddo 12
      enddo 13
      return
      END
```

(Double precision versions of `sprsax` and `sprstx`, named `dsprsax` and `dsprstx`, are used by the routine `atimes` later in this section. You can easily make the conversion, or else get the converted routines from the *Numerical Recipes* diskettes.)

In fact, because the choice of row-indexed storage treats rows and columns quite differently, it is quite an involved operation to construct the transpose of a matrix, given the matrix itself in row-indexed sparse storage mode. When the operation cannot be avoided, it is done as follows: An index of all off-diagonal elements by their columns is constructed (see §8.4). The elements are then written to the output array in column order. As each element is written, its row is determined and stored. Finally, the elements in each column are sorted by row.

```
      SUBROUTINE sprstp(sa,ija,sb,ijb)
      INTEGER ija(*),ijb(*)
      REAL sa(*),sb(*)
C     USES iindexx                   Version of indexx with all REAL variables changed to INTEGER.
        Construct the transpose of a sparse square matrix, from row-index sparse storage arrays sa
        and ija into arrays sb and ijb.
      INTEGER j,jl,jm,jp,ju,k,m,n2,noff,inc,iv
      REAL v
      n2=ija(1)                      Linear size of matrix plus 2.
      do 11 j=1,n2-2                 Diagonal elements.
        sb(j)=sa(j)
      enddo 11
      call iindexx(ija(n2-1)-ija(1),ija(n2),ijb(n2))
        Index all off-diagonal elements by their columns.
      jp=0
      do 13 k=ija(1),ija(n2-1)-1     Loop over output off-diagonal elements.
        m=ijb(k)+n2-1                Use index table to store by (former) columns.
        sb(k)=sa(m)
        do 12 j=jp+1,ija(m)          Fill in the index to any omitted rows.
          ijb(j)=k
        enddo 12
        jp=ija(m)                    Use bisection to find which row element m is in and put that
        jl=1                             into ijb(k).
        ju=n2-1
5       if (ju-jl.gt.1) then
          jm=(ju+jl)/2
          if(ija(jm).gt.m)then
            ju=jm
          else
```

```
                jl=jm
            endif
            goto 5
        endif
        ijb(k)=jl
    enddo 13
    do 14 j=jp+1,n2-1
        ijb(j)=ija(n2-1)
    enddo 14                    Make a final pass to sort each row by Shell sort algorithm.
    do 16 j=1,n2-2
        jl=ijb(j+1)-ijb(j)
        noff=ijb(j)-1
        inc=1
1       inc=3*inc+1
        if(inc.le.jl)goto 1
2       continue
            inc=inc/3
            do 15 k=noff+inc+1,noff+jl
                iv=ijb(k)
                v=sb(k)
                m=k
3               if(ijb(m-inc).gt.iv)then
                    ijb(m)=ijb(m-inc)
                    sb(m)=sb(m-inc)
                    m=m-inc
                    if(m-noff.le.inc)goto 4
                goto 3
                endif
4               ijb(m)=iv
                sb(m)=v
            enddo 15
        if(inc.gt.1)goto 2
    enddo 16
    return
    END
```

The above routine embeds internally a sorting algorithm from §8.1, but calls the external routine `iindexx` to construct the initial column index. This routine is identical to `indexx`, as listed in §8.4, except that the latter's two `REAL` declarations should be changed to `integer`. (The *Numerical Recipes* diskettes include both `indexx` and `iindexx`.) In fact, you can often use `indexx` *without* making these changes, since many computers have the property that numerical values will sort correctly independently of whether they are interpreted as floating or integer values.

As final examples of the manipulation of sparse matrices, we give two routines for the multiplication of two sparse matrices. These are useful for techniques to be described in §13.10.

In general, the product of two sparse matrices is not itself sparse. One therefore wants to limit the size of the product matrix in one of two ways: either compute only those elements of the product that are specified in advance by a known pattern of sparsity, or else compute all nonzero elements, but store only those whose magnitude exceeds some threshold value. The former technique, when it can be used, is quite efficient. The pattern of sparsity is specified by furnishing an index array in row-index sparse storage format (e.g., `ija`). The program then constructs a corresponding value array (e.g., `sa`). The latter technique runs the danger of excessive compute times and unknown output sizes, so it must be used cautiously.

With row-index storage, it is much more natural to multiply a matrix (on the left) by the *transpose* of a matrix (on the right), so that one is crunching rows on rows, rather than rows on columns. Our routines therefore calculate $\mathbf{A} \cdot \mathbf{B}^T$, rather than $\mathbf{A} \cdot \mathbf{B}$. This means that you have to run your right-hand matrix through the transpose routine `sprstp` before sending it to the matrix multiply routine.

The two implementing routines, `sprspm` for "pattern multiply" and `sprstm` for "threshold multiply" are quite similar in structure. Both are complicated by the logic of the various

combinations of diagonal or off-diagonal elements for the two input streams and output stream.

```
      SUBROUTINE sprspm(sa,ija,sb,ijb,sc,ijc)
      INTEGER ija(*),ijb(*),ijc(*)
      REAL sa(*),sb(*),sc(*)
          Matrix multiply A · B^T where A and B are two sparse matrices in row-index storage mode,
          and B^T is the transpose of B. Here, sa and ija store the matrix A; sb and ijb store the
          matrix B. This routine computes only those components of the matrix product that are pre-
          specified by the input index array ijc, which is not modified. On output, the arrays sc and
          ijc give the product matrix in row-index storage mode. For sparse matrix multiplication,
          this routine will often be preceded by a call to sprstp, so as to construct the transpose
          of a known matrix into sb, ijb.
      INTEGER i,ijma,ijmb,j,m,ma,mb,mbb,mn
      REAL sum
      if (ija(1).ne.ijb(1).or.ija(1).ne.ijc(1))
     *     pause 'sprspm sizes do not match'
      do 13 i=1,ijc(1)-2                    Loop over rows.
          j=i                               Set up so that first pass through loop does the diag-
          m=i                                   onal component.
          mn=ijc(i)
          sum=sa(i)*sb(i)
    1     continue                          Main loop over each component to be output.
            mb=ijb(j)
            do 11 ma=ija(i),ija(i+1)-1      Loop through elements in A's row. Convoluted logic,
              ijma=ija(ma)                      following, accounts for the various combinations
              if(ijma.eq.j)then                 of diagonal and off-diagonal elements.
                sum=sum+sa(ma)*sb(j)
              else
    2           if(mb.lt.ijb(j+1))then
                  ijmb=ijb(mb)
                  if(ijmb.eq.i)then
                    sum=sum+sa(i)*sb(mb)
                    mb=mb+1
                    goto 2
                  else if(ijmb.lt.ijma)then
                    mb=mb+1
                    goto 2
                  else if(ijmb.eq.ijma)then
                    sum=sum+sa(ma)*sb(mb)
                    mb=mb+1
                    goto 2
                  endif
                endif
              endif
            enddo 11
            do 12 mbb=mb,ijb(j+1)-1         Exhaust the remainder of B's row.
              if(ijb(mbb).eq.i)then
                sum=sum+sa(i)*sb(mbb)
              endif
            enddo 12
            sc(m)=sum
            sum=0.e0                        Reset indices for next pass through loop.
            if(mn.ge.ijc(i+1))goto 3
            m=mn
            mn=mn+1
            j=ijc(m)
          goto 1
    3     continue
      enddo 13
      return
      END
```

```
      SUBROUTINE sprstm(sa,ija,sb,ijb,thresh,nmax,sc,ijc)
      INTEGER nmax,ija(*),ijb(*),ijc(nmax)
      REAL thresh,sa(*),sb(*),sc(nmax)
```

Matrix multiply $\mathbf{A} \cdot \mathbf{B}^T$ where $\mathbf{A}$ and $\mathbf{B}$ are two sparse matrices in row-index storage mode, and $\mathbf{B}^T$ is the transpose of $\mathbf{B}$. Here, `sa` and `ija` store the matrix $\mathbf{A}$; `sb` and `ijb` store the matrix $\mathbf{B}$. This routine computes all components of the matrix product (which may be non-sparse!), but stores only those whose magnitude exceeds `thresh`. On output, the arrays `sc` and `ijc` (whose maximum size is input as `nmax`) give the product matrix in row-index storage mode. For sparse matrix multiplication, this routine will often be preceded by a call to `sprstp`, so as to construct the transpose of a known matrix into `sb`, `ijb`.

```
      INTEGER i,ijma,ijmb,j,k,ma,mb,mbb
      REAL sum
      if (ija(1).ne.ijb(1)) pause 'sprstm sizes do not match'
      k=ija(1)
      ijc(1)=k
      do 14 i=1,ija(1)-2                    Loop over rows of A,
        do 13 j=1,ijb(1)-2                  and rows of B.
          if(i.eq.j)then
            sum=sa(i)*sb(j)
          else
            sum=0.e0
          endif
          mb=ijb(j)
          do 11 ma=ija(i),ija(i+1)-1        Loop through elements in A's row. Convoluted logic,
            ijma=ija(ma)                        following, accounts for the various combinations
            if(ijma.eq.j)then                   of diagonal and off-diagonal elements.
              sum=sum+sa(ma)*sb(j)
            else
2             if(mb.lt.ijb(j+1))then
                ijmb=ijb(mb)
                if(ijmb.eq.i)then
                  sum=sum+sa(i)*sb(mb)
                  mb=mb+1
                  goto 2
                else if(ijmb.lt.ijma)then
                  mb=mb+1
                  goto 2
                else if(ijmb.eq.ijma)then
                  sum=sum+sa(ma)*sb(mb)
                  mb=mb+1
                  goto 2
                endif
              endif
            endif
          enddo 11
          do 12 mbb=mb,ijb(j+1)-1           Exhaust the remainder of B's row.
            if(ijb(mbb).eq.i)then
              sum=sum+sa(i)*sb(mbb)
            endif
          enddo 12
          if(i.eq.j)then                    Where to put the answer...
            sc(i)=sum
          else if(abs(sum).gt.thresh)then
            if(k.gt.nmax)pause 'sprstm: nmax to small'
            sc(k)=sum
            ijc(k)=j
            k=k+1
          endif
        enddo 13
        ijc(i+1)=k
      enddo 14
      return
      END
```

Conjugate Gradient Method for a Sparse System

So-called *conjugate gradient methods* provide a quite general means for solving the $N \times N$ linear system

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.7.29}$$

The attractiveness of these methods for large sparse systems is that they reference $\mathbf{A}$ only through its multiplication of a vector, or the multiplication of its transpose and a vector. As we have seen, these operations can be very efficient for a properly stored sparse matrix. You, the "owner" of the matrix $\mathbf{A}$, can be asked to provide subroutines that perform these sparse matrix multiplications as efficiently as possible. We, the "grand strategists" supply the general routine, `linbcg` below, that solves the set of linear equations, (2.7.29), using your subroutines.

The simplest, "ordinary" conjugate gradient algorithm [11-13] solves (2.7.29) only in the case that $\mathbf{A}$ is symmetric and positive definite. It is based on the idea of minimizing the function

$$f(\mathbf{x}) = \frac{1}{2}\, \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \cdot \mathbf{x} \tag{2.7.30}$$

This function is minimized when its gradient

$$\nabla f = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{2.7.31}$$

is zero, which is equivalent to (2.7.29). The minimization is carried out by generating a succession of search directions $\mathbf{p}_k$ and improved minimizers $\mathbf{x}_k$. At each stage a quantity α_k is found that minimizes $f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$, and $\mathbf{x}_{k+1}$ is set equal to the new point $\mathbf{x}_k + \alpha_k \mathbf{p}_k$. The $\mathbf{p}_k$ and $\mathbf{x}_k$ are built up in such a way that $\mathbf{x}_{k+1}$ is also the minimizer of f over the whole vector space of directions already taken, $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k\}$. After N iterations you arrive at the minimizer over the entire vector space, i.e., the solution to (2.7.29).

Later, in §10.6, we will generalize this "ordinary" conjugate gradient algorithm to the minimization of arbitrary nonlinear functions. Here, where our interest is in solving linear, but not necessarily positive definite or symmetric, equations, a different generalization is important, the *biconjugate gradient method*. This method does not, in general, have a simple connection with function minimization. It constructs four sequences of vectors, $\mathbf{r}_k$, $\overline{\mathbf{r}}_k$, $\mathbf{p}_k$, $\overline{\mathbf{p}}_k$, $k = 1, 2, \ldots$. You supply the initial vectors $\mathbf{r}_1$ and $\overline{\mathbf{r}}_1$, and set $\mathbf{p}_1 = \mathbf{r}_1$, $\overline{\mathbf{p}}_1 = \overline{\mathbf{r}}_1$. Then you carry out the following recurrence:

$$\begin{aligned}
\alpha_k &= \frac{\overline{\mathbf{r}}_k \cdot \mathbf{r}_k}{\overline{\mathbf{p}}_k \cdot \mathbf{A} \cdot \mathbf{p}_k} \\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k \mathbf{A} \cdot \mathbf{p}_k \\
\overline{\mathbf{r}}_{k+1} &= \overline{\mathbf{r}}_k - \alpha_k \mathbf{A}^T \cdot \overline{\mathbf{p}}_k \\
\beta_k &= \frac{\overline{\mathbf{r}}_{k+1} \cdot \mathbf{r}_{k+1}}{\overline{\mathbf{r}}_k \cdot \mathbf{r}_k} \\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \\
\overline{\mathbf{p}}_{k+1} &= \overline{\mathbf{r}}_{k+1} + \beta_k \overline{\mathbf{p}}_k
\end{aligned} \tag{2.7.32}$$

This sequence of vectors satisfies the *biorthogonality* condition

$$\overline{\mathbf{r}}_i \cdot \mathbf{r}_j = \mathbf{r}_i \cdot \overline{\mathbf{r}}_j = 0, \qquad j < i \tag{2.7.33}$$

and the *biconjugacy* condition

$$\overline{\mathbf{p}}_i \cdot \mathbf{A} \cdot \mathbf{p}_j = \mathbf{p}_i \cdot \mathbf{A}^T \cdot \overline{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.34}$$

There is also a mutual orthogonality,

$$\overline{\mathbf{r}}_i \cdot \mathbf{p}_j = \mathbf{r}_i \cdot \overline{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.35}$$

The proof of these properties proceeds by straightforward induction [14]. As long as the recurrence does not break down earlier because one of the denominators is zero, it must

terminate after $m \le N$ steps with $\mathbf{r}_{m+1} = \overline{\mathbf{r}}_{m+1} = 0$. This is basically because after at most N steps you run out of new orthogonal directions to the vectors you've already constructed.

To use the algorithm to solve the system (2.7.29), make an initial guess $\mathbf{x}_1$ for the solution. Choose $\mathbf{r}_1$ to be the *residual*

$$\mathbf{r}_1 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_1 \tag{2.7.36}$$

and choose $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. Then form the sequence of improved estimates

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{2.7.37}$$

while carrying out the recurrence (2.7.32). Equation (2.7.37) guarantees that $\mathbf{r}_{k+1}$ from the recurrence is in fact the residual $\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{k+1}$ corresponding to $\mathbf{x}_{k+1}$. Since $\mathbf{r}_{m+1} = 0$, $\mathbf{x}_{m+1}$ is the solution to equation (2.7.29).

While there is no guarantee that this whole procedure will not break down or become unstable for general $\mathbf{A}$, in practice this is rare. More importantly, the exact termination in at most N iterations occurs only with exact arithmetic. Roundoff error means that you should regard the process as a genuinely iterative procedure, to be halted when some appropriate error criterion is met.

The ordinary conjugate gradient algorithm is the special case of the biconjugate gradient algorithm when $\mathbf{A}$ is symmetric, and we choose $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. Then $\overline{\mathbf{r}}_k = \mathbf{r}_k$ and $\overline{\mathbf{p}}_k = \mathbf{p}_k$ for all k; you can omit computing them and halve the work of the algorithm. This conjugate gradient version has the interpretation of minimizing equation (2.7.30). If $\mathbf{A}$ is positive definite as well as symmetric, the algorithm cannot break down (in theory!). The routine `linbcg` below indeed reduces to the ordinary conjugate gradient method if you input a symmetric $\mathbf{A}$, but it does all the redundant computations.

Another variant of the general algorithm corresponds to a symmetric but non-positive definite $\mathbf{A}$, with the choice $\overline{\mathbf{r}}_1 = \mathbf{A} \cdot \mathbf{r}_1$ instead of $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. In this case $\overline{\mathbf{r}}_k = \mathbf{A} \cdot \mathbf{r}_k$ and $\overline{\mathbf{p}}_k = \mathbf{A} \cdot \mathbf{p}_k$ for all k. This algorithm is thus equivalent to the ordinary conjugate gradient algorithm, but with all dot products $\mathbf{a} \cdot \mathbf{b}$ replaced by $\mathbf{a} \cdot \mathbf{A} \cdot \mathbf{b}$. It is called the *minimum residual* algorithm, because it corresponds to successive minimizations of the function

$$\Phi(\mathbf{x}) = \frac{1}{2}\,\mathbf{r} \cdot \mathbf{r} = \frac{1}{2}\,|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2 \tag{2.7.38}$$

where the successive iterates $\mathbf{x}_k$ minimize Φ over the same set of search directions $\mathbf{p}_k$ generated in the conjugate gradient method. This algorithm has been generalized in various ways for unsymmetric matrices. The *generalized minimum residual* method (GMRES; see [9,15]) is probably the most robust of these methods.

Note that equation (2.7.38) gives

$$\nabla\Phi(\mathbf{x}) = \mathbf{A}^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \tag{2.7.39}$$

For any nonsingular matrix $\mathbf{A}$, $\mathbf{A}^T \cdot \mathbf{A}$ is symmetric and positive definite. You might therefore be tempted to solve equation (2.7.29) by applying the ordinary conjugate gradient algorithm to the problem

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = \mathbf{A}^T \cdot \mathbf{b} \tag{2.7.40}$$

Don't! The condition number of the matrix $\mathbf{A}^T \cdot \mathbf{A}$ is the square of the condition number of $\mathbf{A}$ (see §2.6 for definition of condition number). A large condition number both increases the number of iterations required, and limits the accuracy to which a solution can be obtained. It is almost always better to apply the biconjugate gradient method to the original matrix $\mathbf{A}$.

So far we have said nothing about the *rate* of convergence of these methods. The ordinary conjugate gradient method works well for matrices that are well-conditioned, i.e., "close" to the identity matrix. This suggests applying these methods to the *preconditioned* form of equation (2.7.29),

$$(\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{A}) \cdot \mathbf{x} = \widetilde{\mathbf{A}}^{-1} \cdot \mathbf{b} \tag{2.7.41}$$

The idea is that you might already be able to solve your linear system easily for some $\widetilde{\mathbf{A}}$ close to $\mathbf{A}$, in which case $\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{A} \approx \mathbf{1}$, allowing the algorithm to converge in fewer steps. The

matrix $\tilde{\mathbf{A}}$ is called a *preconditioner* [11], and the overall scheme given here is known as the *preconditioned biconjugate gradient method* or *PBCG*.

For efficient implementation, the PBCG algorithm introduces an additional set of vectors $\mathbf{z}_k$ and $\overline{\mathbf{z}}_k$ defined by

$$\tilde{\mathbf{A}} \cdot \mathbf{z}_k = \mathbf{r}_k \qquad \text{and} \qquad \tilde{\mathbf{A}}^T \cdot \overline{\mathbf{z}}_k = \overline{\mathbf{r}}_k \tag{2.7.42}$$

and modifies the definitions of α_k, β_k, $\mathbf{p}_k$, and $\overline{\mathbf{p}}_k$ in equation (2.7.32):

$$\begin{aligned} \alpha_k &= \frac{\overline{\mathbf{r}}_k \cdot \mathbf{z}_k}{\overline{\mathbf{p}}_k \cdot \mathbf{A} \cdot \mathbf{p}_k} \\ \beta_k &= \frac{\overline{\mathbf{r}}_{k+1} \cdot \mathbf{z}_{k+1}}{\overline{\mathbf{r}}_k \cdot \mathbf{z}_k} \\ \mathbf{p}_{k+1} &= \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k \\ \overline{\mathbf{p}}_{k+1} &= \overline{\mathbf{z}}_{k+1} + \beta_k \overline{\mathbf{p}}_k \end{aligned} \tag{2.7.43}$$

For `linbcg`, below, we will ask you to supply routines that solve the auxiliary linear systems (2.7.42). If you have no idea what to use for the preconditioner $\tilde{\mathbf{A}}$, then use the diagonal part of $\mathbf{A}$, or even the identity matrix, in which case the burden of convergence will be entirely on the biconjugate gradient method itself.

The routine `linbcg`, below, is based on a program originally written by Anne Greenbaum. (See [13] for a different, less sophisticated, implementation.) There are a few wrinkles you should know about.

What constitutes "good" convergence is rather application dependent. The routine `linbcg` therefore provides for four possibilities, selected by setting the flag `itol` on input. If `itol=1`, iteration stops when the quantity $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|/|\mathbf{b}|$ is less than the input quantity `tol`. If `itol=2`, the required criterion is

$$|\tilde{\mathbf{A}}^{-1} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b})|/|\tilde{\mathbf{A}}^{-1} \cdot \mathbf{b}| < \texttt{tol} \tag{2.7.44}$$

If `itol=3`, the routine uses its own estimate of the error in $\mathbf{x}$, and requires its magnitude, divided by the magnitude of $\mathbf{x}$, to be less than `tol`. The setting `itol=4` is the same as `itol=3`, except that the largest (in absolute value) component of the error and largest component of $\mathbf{x}$ are used instead of the vector magnitude (that is, the L_∞ norm instead of the L_2 norm). You may need to experiment to find which of these convergence criteria is best for your problem.

On output, `err` is the tolerance actually achieved. If the returned count `iter` does not indicate that the maximum number of allowed iterations `itmax` was exceeded, then `err` should be less than `tol`. If you want to do further iterations, leave all returned quantities as they are and call the routine again. The routine loses its memory of the spanned conjugate gradient subspace between calls, however, so you should not force it to return more often than about every N iterations.

Finally, note that `linbcg` is furnished in double precision, since it will be usually be used when N is quite large.

```
      SUBROUTINE linbcg(n,b,x,itol,tol,itmax,iter,err)
      INTEGER iter,itmax,itol,n,NMAX
      DOUBLE PRECISION err,tol,b(*),x(*),EPS
      PARAMETER (NMAX=1024,EPS=1.d-14)
```

Double precision is a good idea in this routine.

```
C     USES atimes,asolve,snrm
```

Solves $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for `x(1:n)`, given `b(1:n)`, by the iterative biconjugate gradient method. On input `x(1:n)` should be set to an initial guess of the solution (or all zeros); `itol` is 1,2,3, or 4, specifying which convergence test is applied (see text); `itmax` is the maximum number of allowed iterations; and `tol` is the desired convergence tolerance. On output, `x(1:n)` is reset to the improved solution, `iter` is the number of iterations actually taken, and `err` is the estimated error. The matrix $\mathbf{A}$ is referenced only through the user-supplied routines `atimes`, which computes the product of either $\mathbf{A}$ or its transpose on a vector; and `asolve`, which solves $\tilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$ or $\tilde{\mathbf{A}}^T \cdot \mathbf{x} = \mathbf{b}$ for some preconditioner matrix $\tilde{\mathbf{A}}$ (possibly the trivial diagonal part of $\mathbf{A}$).

```
      INTEGER j
```

```
      DOUBLE PRECISION ak,akden,bk,bkden,bknum,bnrm,dxnrm,
     *      xnrm,zm1nrm,znrm,p(NMAX),pp(NMAX),r(NMAX),rr(NMAX),
     *      z(NMAX),zz(NMAX),snrm
      iter=0                                    Calculate initial residual.
      call atimes(n,x,r,0)                      Input to atimes is x(1:n), output is r(1:n);
      do 11 j=1,n                                   the final 0 indicates that the matrix (not
        r(j)=b(j)-r(j)                              its transpose) is to be used.
        rr(j)=r(j)
      enddo 11
C     call atimes(n,r,rr,0)                     Uncomment this line to get the "minimum
      if(itol.eq.1) then                            residual" variant of the algorithm.
        bnrm=snrm(n,b,itol)
        call asolve(n,r,z,0)                    Input to asolve is r(1:n), output is z(1:n);
      else if (itol.eq.2) then                      the final 0 indicates that the matrix Ã
        call asolve(n,b,z,0)                        (not its transpose) is to be used.
        bnrm=snrm(n,z,itol)
        call asolve(n,r,z,0)
      else if (itol.eq.3.or.itol.eq.4) then
        call asolve(n,b,z,0)
        bnrm=snrm(n,z,itol)
        call asolve(n,r,z,0)
        znrm=snrm(n,z,itol)
      else
        pause 'illegal itol in linbcg'
      endif
100   if (iter.le.itmax) then                   Main loop.
        iter=iter+1
        call asolve(n,rr,zz,1)                  Final 1 indicates use of transpose matrix Ã^T.
        bknum=0.d0
        do 12 j=1,n                             Calculate coefficient bk and direction vectors
          bknum=bknum+z(j)*rr(j)                    p and pp.
        enddo 12
        if(iter.eq.1) then
          do 13 j=1,n
            p(j)=z(j)
            pp(j)=zz(j)
          enddo 13
        else
          bk=bknum/bkden
          do 14 j=1,n
            p(j)=bk*p(j)+z(j)
            pp(j)=bk*pp(j)+zz(j)
          enddo 14
        endif
        bkden=bknum                             Calculate coefficient ak, new iterate x, and
        call atimes(n,p,z,0)                        new residuals r and rr.
        akden=0.d0
        do 15 j=1,n
          akden=akden+z(j)*pp(j)
        enddo 15
        ak=bknum/akden
        call atimes(n,pp,zz,1)
        do 16 j=1,n
          x(j)=x(j)+ak*p(j)
          r(j)=r(j)-ak*z(j)
          rr(j)=rr(j)-ak*zz(j)
        enddo 16
        call asolve(n,r,z,0)                    Solve Ã·z = r and check stopping criterion.
        if(itol.eq.1)then
          err=snrm(n,r,itol)/bnrm
        else if(itol.eq.2)then
          err=snrm(n,z,itol)/bnrm
        else if(itol.eq.3.or.itol.eq.4)then
          zm1nrm=znrm
```

```
        znrm=snrm(n,z,itol)
        if(abs(zm1nrm-znrm).gt.EPS*znrm) then
            dxnrm=abs(ak)*snrm(n,p,itol)
            err=znrm/abs(zm1nrm-znrm)*dxnrm
        else
            err=znrm/bnrm                    Error may not be accurate, so loop again.
            goto 100
        endif
        xnrm=snrm(n,x,itol)
        if(err.le.0.5d0*xnrm) then
            err=err/xnrm
        else
            err=znrm/bnrm                    Error may not be accurate, so loop again.
            goto 100
        endif
    endif
    write (*,*) ' iter=',iter,' err=',err
if(err.gt.tol) goto 100
endif
return
END
```

The routine `linbcg` uses this short utility for computing vector norms:

```
FUNCTION snrm(n,sx,itol)
INTEGER n,itol,i,isamax
DOUBLE PRECISION sx(n),snrm
    Compute one of two norms for a vector sx(1:n), as signaled by itol. Used by linbcg.
if (itol.le.3)then
    snrm=0.
    do 11 i=1,n                  Vector magnitude norm.
        snrm=snrm+sx(i)**2
    enddo 11
    snrm=sqrt(snrm)
else
    isamax=1
    do 12 i=1,n                  Largest component norm.
        if(abs(sx(i)).gt.abs(sx(isamax))) isamax=i
    enddo 12
    snrm=abs(sx(isamax))
endif
return
END
```

So that the specifications for the routines `atimes` and `asolve` are clear, we list here simple versions that assume a matrix **A** stored somewhere in row-index sparse format.

```
SUBROUTINE atimes(n,x,r,itrnsp)
INTEGER n,itrnsp,ija,NMAX
DOUBLE PRECISION x(n),r(n),sa
PARAMETER (NMAX=1000)
COMMON /mat/ sa(NMAX),ija(NMAX)     The matrix is stored somewhere.
C   USES dsprsax,dsprstx            DOUBLE PRECISION versions of sprsax and sprstx.
if (itrnsp.eq.0) then
    call dsprsax(sa,ija,x,r,n)
else
    call dsprstx(sa,ija,x,r,n)
endif
return
END
```

```
SUBROUTINE asolve(n,b,x,itrnsp)
INTEGER n,itrnsp,ija,NMAX,i
DOUBLE PRECISION x(n),b(n),sa
PARAMETER (NMAX=1000)
COMMON /mat/ sa(NMAX),ija(NMAX)
do 11 i=1,n
    x(i)=b(i)/sa(i)
enddo 11
return
END
```

The matrix is stored somewhere.

The matrix $\widetilde{\mathbf{A}}$ is the diagonal part of $\mathbf{A}$, stored in the first n elements of sa. Since the transpose matrix has the same diagonal, the flag itrnsp is not used.

CITED REFERENCES AND FURTHER READING:

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter I.3 (by J.K. Reid). [2]

George, A., and Liu, J.W.H. 1981, *Computer Solution of Large Sparse Positive Definite Systems* (Englewood Cliffs, NJ: Prentice-Hall). [3]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.). [4]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [5]

Eisenstat, S.C., Gursky, M.C., Schultz, M.H., and Sherman, A.H. 1977, *Yale Sparse Matrix Package*, Technical Reports 112 and 114 (Yale University Department of Computer Science). [6]

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.2.6. [7]

Kincaid, D.R., Respess, J.R., Young, D.M., and Grimes, R.G. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 302–322. [8]

PCGPAK User's Guide (New Haven: Scientific Computing Associates, Inc.). [9]

Bentley, J. 1986, *Programming Pearls* (Reading, MA: Addison-Wesley), §9. [10]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapters 4 and 10, particularly §§10.2–10.3. [11]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 8. [12]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill). [13]

Fletcher, R. 1976, in *Numerical Analysis Dundee 1975*, Lecture Notes in Mathematics, vol. 506, A. Dold and B Eckmann, eds. (Berlin: Springer-Verlag), pp. 73–89. [14]

Saad, Y., and Schulz, M. 1986, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856–869. [15]

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press).

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.).

2.8 Vandermonde Matrices and Toeplitz Matrices

In §2.4 the case of a tridiagonal matrix was treated specially, because that particular type of linear system admits a solution in only of order N operations, rather than of order N^3 for the general linear problem. When such particular types

```
SUBROUTINE asolve(n,b,x,itrnsp)
INTEGER n,itrnsp,ija,NMAX,i
DOUBLE PRECISION x(n),b(n),sa
PARAMETER (NMAX=1000)
COMMON /mat/ sa(NMAX),ija(NMAX)
do 11 i=1,n
    x(i)=b(i)/sa(i)
enddo 11
return
END
```

The matrix is stored somewhere.

The matrix $\widetilde{\mathbf{A}}$ is the diagonal part of $\mathbf{A}$, stored in the first n elements of sa. Since the transpose matrix has the same diagonal, the flag itrnsp is not used.

CITED REFERENCES AND FURTHER READING:

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter I.3 (by J.K. Reid). [2]

George, A., and Liu, J.W.H. 1981, *Computer Solution of Large Sparse Positive Definite Systems* (Englewood Cliffs, NJ: Prentice-Hall). [3]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.). [4]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [5]

Eisenstat, S.C., Gursky, M.C., Schultz, M.H., and Sherman, A.H. 1977, *Yale Sparse Matrix Package*, Technical Reports 112 and 114 (Yale University Department of Computer Science). [6]

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.2.6. [7]

Kincaid, D.R., Respess, J.R., Young, D.M., and Grimes, R.G. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 302–322. [8]

PCGPAK User's Guide (New Haven: Scientific Computing Associates, Inc.). [9]

Bentley, J. 1986, *Programming Pearls* (Reading, MA: Addison-Wesley), §9. [10]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapters 4 and 10, particularly §§10.2–10.3. [11]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 8. [12]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill). [13]

Fletcher, R. 1976, in *Numerical Analysis Dundee 1975*, Lecture Notes in Mathematics, vol. 506, A. Dold and B Eckmann, eds. (Berlin: Springer-Verlag), pp. 73–89. [14]

Saad, Y., and Schulz, M. 1986, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856–869. [15]

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press).

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.).

2.8 Vandermonde Matrices and Toeplitz Matrices

In §2.4 the case of a tridiagonal matrix was treated specially, because that particular type of linear system admits a solution in only of order N operations, rather than of order N^3 for the general linear problem. When such particular types

exist, it is important to know about them. Your computational savings, should you ever happen to be working on a problem that involves the right kind of particular type, can be enormous.

This section treats two special types of matrices that can be solved in of order N^2 operations, not as good as tridiagonal, but a lot better than the general case. (Other than the operations count, these two types having nothing in common.) Matrices of the first type, termed *Vandermonde matrices*, occur in some problems having to do with the fitting of polynomials, the reconstruction of distributions from their moments, and also other contexts. In this book, for example, a Vandermonde problem crops up in §3.5. Matrices of the second type, termed *Toeplitz matrices*, tend to occur in problems involving deconvolution and signal processing. In this book, a Toeplitz problem is encountered in §13.7.

These are not the *only* special types of matrices worth knowing about. The *Hilbert matrices*, whose components are of the form $a_{ij} = 1/(i + j - 1),\ i, j = 1, \ldots, N$ can be inverted by an exact integer algorithm, and are very *difficult* to invert in any other way, since they are notoriously ill-conditioned (see [1] for details). The Sherman-Morrison and Woodbury formulas, discussed in §2.7, can sometimes be used to convert new special forms into old ones. Reference [2] gives some other special forms. We have not found these additional forms to arise as frequently as the two that we now discuss.

Vandermonde Matrices

A Vandermonde matrix of size $N \times N$ is completely determined by N arbitrary numbers $x_1, x_2, \ldots, x_N$, in terms of which its N^2 components are the integer powers $x_i^{j-1},\ i, j = 1, \ldots, N$. Evidently there are two possible such forms, depending on whether we view the i's as rows, j's as columns, or vice versa. In the former case, we get a linear system of equations that looks like this,

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.8.1}$$

Performing the matrix multiplication, you will see that this equation solves for the unknown coefficients c_i which fit a polynomial to the N pairs of abscissas and ordinates (x_j, y_j). Precisely this problem will arise in §3.5, and the routine given there will solve (2.8.1) by the method that we are about to describe.

The alternative identification of rows and columns leads to the set of equations

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ & & \cdots & \\ x_1^{N-1} & x_2^{N-1} & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_N \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \cdots \\ q_N \end{bmatrix} \tag{2.8.2}$$

Write this out and you will see that it relates to the *problem of moments*: Given the values of N points x_i, find the unknown weights w_i, assigned so as to match the given values q_j of the first N moments. (For more on this problem, consult [3].) The routine given in this section solves (2.8.2).

The method of solution of both (2.8.1) and (2.8.2) is closely related to Lagrange's polynomial interpolation formula, which we will not formally meet until §3.1 below. Notwithstanding, the following derivation should be comprehensible:

Let $P_j(x)$ be the polynomial of degree $N-1$ defined by

$$P_j(x) = \prod_{\substack{n=1 \\ (n\neq j)}}^{N} \frac{x - x_n}{x_j - x_n} = \sum_{k=1}^{N} A_{jk} x^{k-1} \tag{2.8.3}$$

Here the meaning of the last equality is to define the components of the matrix A_{ij} as the coefficients that arise when the product is multiplied out and like terms collected.

The polynomial $P_j(x)$ is a function of x generally. But you will notice that it is specifically designed so that it takes on a value of zero at all x_i with $i \neq j$, and has a value of unity at $x = x_j$. In other words,

$$P_j(x_i) = \delta_{ij} = \sum_{k=1}^{N} A_{jk} x_i^{k-1} \tag{2.8.4}$$

But (2.8.4) says that A_{jk} is exactly the inverse of the matrix of components x_i^{k-1}, which appears in (2.8.2), with the subscript as the column index. Therefore the solution of (2.8.2) is just that matrix inverse times the right-hand side,

$$w_j = \sum_{k=1}^{N} A_{jk} q_k \tag{2.8.5}$$

As for the transpose problem (2.8.1), we can use the fact that the inverse of the transpose is the transpose of the inverse, so

$$c_j = \sum_{k=1}^{N} A_{kj} y_k \tag{2.8.6}$$

The routine in §3.5 implements this.

It remains to find a good way of multiplying out the monomial terms in (2.8.3), in order to get the components of A_{jk}. This is essentially a bookkeeping problem, and we will let you read the routine itself to see how it can be solved. One trick is to define a master $P(x)$ by

$$P(x) \equiv \prod_{n=1}^{N} (x - x_n) \tag{2.8.7}$$

work out its coefficients, and then obtain the numerators and denominators of the specific P_j's via synthetic division by the one supernumerary term. (See §5.3 for more on synthetic division.) Since each such division is only a process of order N, the total procedure is of order N^2.

You should be warned that Vandermonde systems are notoriously ill-conditioned, by their very nature. (As an aside anticipating §5.8, the reason is the same as that which makes Chebyshev fitting so impressively accurate: there exist high-order polynomials that are very good uniform fits to zero. Hence roundoff error can introduce rather substantial coefficients of the leading terms of these polynomials.) It is a good idea always to compute Vandermonde problems in double precision.

The routine for (2.8.2) which follows is due to G.B. Rybicki.

```
SUBROUTINE vander(x,w,q,n)
INTEGER n,NMAX
DOUBLE PRECISION q(n),w(n),x(n)
PARAMETER (NMAX=100)
```

Solves the Vandermonde linear system $\sum_{i=1}^{N} x_i^{k-1} w_i = q_k$ $(k = 1, \ldots, N)$. Input consists of the vectors `x(1:n)` and `q(1:n)`; the vector `w(1:n)` is output.

```
    Parameters: NMAX is the maximum expected value of n.
INTEGER i,j,k
DOUBLE PRECISION b,s,t,xx,c(NMAX)
if(n.eq.1)then
    w(1)=q(1)
else
    do 11 i=1,n                    Initialize array.
        c(i)=0.d0
    enddo 11
    c(n)=-x(1)                     Coefficients of the master polynomial are found by recur-
    do 13 i=2,n                        sion.
        xx=-x(i)
        do 12 j=n+1-i,n-1
            c(j)=c(j)+xx*c(j+1)
        enddo 12
        c(n)=c(n)+xx
    enddo 13
    do 15 i=1,n                    Each subfactor in turn
        xx=x(i)
        t=1.d0
        b=1.d0
        s=q(n)
        do 14 k=n,2,-1             is synthetically divided,
            b=c(k)+xx*b
            s=s+q(k-1)*b           matrix-multiplied by the right-hand side,
            t=xx*t+b
        enddo 14
        w(i)=s/t                   and supplied with a denominator.
    enddo 15
endif
return
END
```

Toeplitz Matrices

An $N \times N$ Toeplitz matrix is specified by giving $2N-1$ numbers R_k, $k = -N+1, \ldots, -1, 0, 1, \ldots, N-1$. Those numbers are then emplaced as matrix elements constant along the (upper-left to lower-right) diagonals of the matrix:

$$\begin{bmatrix} R_0 & R_{-1} & R_{-2} & \cdots & R_{-(N-2)} & R_{-(N-1)} \\ R_1 & R_0 & R_{-1} & \cdots & R_{-(N-3)} & R_{-(N-2)} \\ R_2 & R_1 & R_0 & \cdots & R_{-(N-4)} & R_{-(N-3)} \\ \cdots & & & \cdots & & \\ R_{N-2} & R_{N-3} & R_{N-4} & \cdots & R_0 & R_{-1} \\ R_{N-1} & R_{N-2} & R_{N-3} & \cdots & R_1 & R_0 \end{bmatrix} \tag{2.8.8}$$

The linear Toeplitz problem can thus be written as

$$\sum_{j=1}^{N} R_{i-j} x_j = y_i \qquad (i = 1, \ldots, N) \tag{2.8.9}$$

where the x_j's, $j = 1, \ldots, N$, are the unknowns to be solved for.

The Toeplitz matrix is symmetric if $R_k = R_{-k}$ for all k. Levinson [4] developed an algorithm for fast solution of the symmetric Toeplitz problem, by a *bordering method*, that is, a recursive procedure that solves the M-dimensional Toeplitz problem

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad (i = 1, \ldots, M) \tag{2.8.10}$$

in turn for $M = 1, 2, \ldots$ until $M = N$, the desired result, is finally reached. The vector $x_j^{(M)}$ is the result at the Mth stage, and becomes the desired answer only when N is reached.

Levinson's method is well documented in standard texts (e.g., [5]). The useful fact that the method generalizes to the *nonsymmetric* case seems to be less well known. At some risk of excessive detail, we therefore give a derivation here, due to G.B. Rybicki.

In following a recursion from step M to step $M+1$ we find that our developing solution $x^{(M)}$ changes in this way:

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad i = 1, \ldots, M \tag{2.8.11}$$

becomes

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M+1)} + R_{i-(M+1)} x_{M+1}^{(M+1)} = y_i \qquad i = 1, \ldots, M+1 \tag{2.8.12}$$

By eliminating y_i we find

$$\sum_{j=1}^{M} R_{i-j} \left(\frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \right) = R_{i-(M+1)} \qquad i = 1, \ldots, M \tag{2.8.13}$$

or by letting $i \to M+1-i$ and $j \to M+1-j$,

$$\sum_{j=1}^{M} R_{j-i} G_j^{(M)} = R_{-i} \tag{2.8.14}$$

where

$$G_j^{(M)} \equiv \frac{x_{M+1-j}^{(M)} - x_{M+1-j}^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.15}$$

To put this another way,

$$x_{M+1-j}^{(M+1)} = x_{M+1-j}^{(M)} - x_{M+1}^{(M+1)} G_j^{(M)} \qquad j = 1, \ldots, M \tag{2.8.16}$$

Thus, if we can use recursion to find the order M quantities $x^{(M)}$ and $G^{(M)}$ *and* the single order $M+1$ quantity $x_{M+1}^{(M+1)}$, then all of the other $x_j^{(M+1)}$ will follow. Fortunately, the quantity $x_{M+1}^{(M+1)}$ follows from equation (2.8.12) with $i = M+1$,

$$\sum_{j=1}^{M} R_{M+1-j} x_j^{(M+1)} + R_0 x_{M+1}^{(M+1)} = y_{M+1} \tag{2.8.17}$$

For the unknown order $M+1$ quantities $x_j^{(M+1)}$ we can substitute the previous order quantities in G since

$$G_{M+1-j}^{(M)} = \frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.18}$$

The result of this operation is

$$x_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{M+1-j} x_j^{(M)} - y_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G_{M+1-j}^{(M)} - R_0} \tag{2.8.19}$$

The only remaining problem is to develop a recursion relation for G. Before we do that, however, we should point out that there are actually two distinct sets of solutions to the original linear problem for a nonsymmetric matrix, namely right-hand solutions (which we

have been discussing) and left-hand solutions z_i. The formalism for the left-hand solutions differs only in that we deal with the equations

$$\sum_{j=1}^{M} R_{j-i} z_j^{(M)} = y_i \qquad i = 1, \ldots, M \tag{2.8.20}$$

Then, the same sequence of operations on this set leads to

$$\sum_{j=1}^{M} R_{i-j} H_j^{(M)} = R_i \tag{2.8.21}$$

where

$$H_j^{(M)} \equiv \frac{z_{M+1-j}^{(M)} - z_{M+1-j}^{(M+1)}}{z_{M+1}^{(M+1)}} \tag{2.8.22}$$

(compare with 2.8.14 – 2.8.15). The reason for mentioning the left-hand solutions now is that, by equation (2.8.21), the H_j satisfy exactly the same equation as the x_j except for the substitution $y_i \to R_i$ on the right-hand side. Therefore we can quickly deduce from equation (2.8.19) that

$$H_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{M+1-j} H_j^{(M)} - R_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G_{M+1-j}^{(M)} - R_0} \tag{2.8.23}$$

By the same token, G satisfies the same equation as z, except for the substitution $y_i \to R_{-i}$. This gives

$$G_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{j-M-1} G_j^{(M)} - R_{-M-1}}{\sum_{j=1}^{M} R_{j-M-1} H_{M+1-j}^{(M)} - R_0} \tag{2.8.24}$$

The same "morphism" also turns equation (2.8.16), and its partner for z, into the final equations

$$\begin{aligned} G_j^{(M+1)} &= G_j^{(M)} - G_{M+1}^{(M+1)} H_{M+1-j}^{(M)} \\ H_j^{(M+1)} &= H_j^{(M)} - H_{M+1}^{(M+1)} G_{M+1-j}^{(M)} \end{aligned} \tag{2.8.25}$$

Now, starting with the initial values

$$x_1^{(1)} = y_1/R_0 \qquad G_1^{(1)} = R_{-1}/R_0 \qquad H_1^{(1)} = R_1/R_0 \tag{2.8.26}$$

we can recurse away. At each stage M we use equations (2.8.23) and (2.8.24) to find $H_{M+1}^{(M+1)}, G_{M+1}^{(M+1)}$, and then equation (2.8.25) to find the other components of $H^{(M+1)}, G^{(M+1)}$. From there the vectors $x^{(M+1)}$ and/or $z^{(M+1)}$ are easily calculated.

The program below does this. It incorporates the second equation in (2.8.25) in the form

$$H_{M+1-j}^{(M+1)} = H_{M+1-j}^{(M)} - H_{M+1}^{(M+1)} G_j^{(M)} \tag{2.8.27}$$

so that the computation can be done "in place."

Notice that the above algorithm fails if $R_0 = 0$. In fact, because the bordering method does not allow pivoting, the algorithm will fail if any of the diagonal principal minors of the original Toeplitz matrix vanish. (Compare with discussion of the tridiagonal algorithm in §2.4.) If the algorithm fails, your matrix is not necessarily singular — you might just have to solve your problem by a slower and more general algorithm such as LU decomposition with pivoting.

The routine that implements equations (2.8.23)–(2.8.27) is also due to Rybicki. Note that the routine's `r(n+j)` is equal to R_j above, so that subscripts on the `r` array vary from 1 to $2N - 1$.

```
      SUBROUTINE toeplz(r,x,y,n)
      INTEGER n,NMAX
      REAL r(2*n-1),x(n),y(n)
      PARAMETER (NMAX=100)
        Solves the Toeplitz system sum_{j=1}^{N} R_{(N+i-j)} x_j = y_i (i = 1,...,N). The Toeplitz matrix
        need not be symmetric. y and r are input arrays of length n and 2*n-1, respectively. x
        is the output array, of length n.
        Parameter: NMAX is the maximum anticipated value of n.
      INTEGER j,k,m,m1,m2
      REAL pp,pt1,pt2,qq,qt1,qt2,sd,sgd,sgn,shn,sxn,
     *     g(NMAX),h(NMAX)
      if(r(n).eq.0.) goto 99
      x(1)=y(1)/r(n)                  Initialize for the recursion.
      if(n.eq.1)return
      g(1)=r(n-1)/r(n)
      h(1)=r(n+1)/r(n)
      do 15 m=1,n                     Main loop over the recursion.
        m1=m+1
        sxn=-y(m1)                    Compute numerator and denominator for x,
        sd=-r(n)
        do 11 j=1,m
          sxn=sxn+r(n+m1-j)*x(j)
          sd=sd+r(n+m1-j)*g(m-j+1)
        enddo 11
        if(sd.eq.0.)goto 99
        x(m1)=sxn/sd                  whence x.
        do 12 j=1,m
          x(j)=x(j)-x(m1)*g(m-j+1)
        enddo 12
        if(m1.eq.n)return
        sgn=-r(n-m1)                  Compute numerator and denominator for G and H,
        shn=-r(n+m1)
        sgd=-r(n)
        do 13 j=1,m
          sgn=sgn+r(n+j-m1)*g(j)
          shn=shn+r(n+m1-j)*h(j)
          sgd=sgd+r(n+j-m1)*h(m-j+1)
        enddo 13
        if(sd.eq.0..or.sgd.eq.0.)goto 99
        g(m1)=sgn/sgd                 whence G and H.
        h(m1)=shn/sd
        k=m
        m2=(m+1)/2
        pp=g(m1)
        qq=h(m1)
        do 14 j=1,m2
          pt1=g(j)
          pt2=g(k)
          qt1=h(j)
          qt2=h(k)
          g(j)=pt1-pp*qt2
          g(k)=pt2-pp*qt1
          h(j)=qt1-qq*pt2
          h(k)=qt2-qq*pt1
          k=k-1
        enddo 14
      enddo 15                        Back for another recurrence.
      pause 'never get here in toeplz'
   99 pause 'singular principal minor in toeplz'
      END
```

If you are in the business of solving *very* large Toeplitz systems, you should find out about so-called "new, fast" algorithms, which require only on the order of $N(\log N)^2$ operations,

compared to N^2 for Levinson's method. These methods are too complicated to include here. Papers by Bunch [6] and de Hoog [7] will give entry to the literature.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 5 [also treats some other special forms].

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), §19. [1]

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley). [2]

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), pp. 394ff. [3]

Levinson, N., Appendix B of N. Wiener, 1949, *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (New York: Wiley). [4]

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall), pp. 163ff. [5]

Bunch, J.R. 1985, *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 349–364. [6]

de Hoog, F. 1987, *Linear Algebra and Its Applications*, vol. 88/89, pp. 123–138. [7]

2.9 Cholesky Decomposition

If a square matrix $\mathbf{A}$ happens to be symmetric and positive definite, then it has a special, more efficient, triangular decomposition. *Symmetric* means that $a_{ij} = a_{ji}$ for $i, j = 1, \ldots, N$, while *positive definite* means that

$$\mathbf{v} \cdot \mathbf{A} \cdot \mathbf{v} > 0 \quad \text{for all vectors } \mathbf{v} \tag{2.9.1}$$

(In Chapter 11 we will see that positive definite has the equivalent interpretation that $\mathbf{A}$ has all positive eigenvalues.) While symmetric, positive definite matrices are rather special, they occur quite frequently in some applications, so their special factorization, called *Cholesky decomposition*, is good to know about. When you can use it, Cholesky decomposition is about a factor of two faster than alternative methods for solving linear equations.

Instead of seeking arbitrary lower and upper triangular factors $\mathbf{L}$ and $\mathbf{U}$, Cholesky decomposition constructs a lower triangular matrix $\mathbf{L}$ whose transpose $\mathbf{L}^T$ can itself serve as the upper triangular part. In other words we replace equation (2.3.1) by

$$\mathbf{L} \cdot \mathbf{L}^T = \mathbf{A} \tag{2.9.2}$$

This factorization is sometimes referred to as "taking the square root" of the matrix $\mathbf{A}$. The components of $\mathbf{L}^T$ are of course related to those of $\mathbf{L}$ by

$$L^T_{ij} = L_{ji} \tag{2.9.3}$$

Writing out equation (2.9.2) in components, one readily obtains the analogs of equations (2.3.12)–(2.3.13),

$$L_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \right)^{1/2} \tag{2.9.4}$$

and

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right) \qquad j = i+1, i+2, \ldots, N \tag{2.9.5}$$

compared to N^2 for Levinson's method. These methods are too complicated to include here. Papers by Bunch [6] and de Hoog [7] will give entry to the literature.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 5 [also treats some other special forms].

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), §19. [1]

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley). [2]

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), pp. 394ff. [3]

Levinson, N., Appendix B of N. Wiener, 1949, *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (New York: Wiley). [4]

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall), pp. 163ff. [5]

Bunch, J.R. 1985, *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 349–364. [6]

de Hoog, F. 1987, *Linear Algebra and Its Applications*, vol. 88/89, pp. 123–138. [7]

2.9 Cholesky Decomposition

If a square matrix $\mathbf{A}$ happens to be symmetric and positive definite, then it has a special, more efficient, triangular decomposition. *Symmetric* means that $a_{ij} = a_{ji}$ for $i, j = 1, \ldots, N$, while *positive definite* means that

$$\mathbf{v} \cdot \mathbf{A} \cdot \mathbf{v} > 0 \quad \text{for all vectors } \mathbf{v} \tag{2.9.1}$$

(In Chapter 11 we will see that positive definite has the equivalent interpretation that $\mathbf{A}$ has all positive eigenvalues.) While symmetric, positive definite matrices are rather special, they occur quite frequently in some applications, so their special factorization, called *Cholesky decomposition*, is good to know about. When you can use it, Cholesky decomposition is about a factor of two faster than alternative methods for solving linear equations.

Instead of seeking arbitrary lower and upper triangular factors $\mathbf{L}$ and $\mathbf{U}$, Cholesky decomposition constructs a lower triangular matrix $\mathbf{L}$ whose transpose $\mathbf{L}^T$ can itself serve as the upper triangular part. In other words we replace equation (2.3.1) by

$$\mathbf{L} \cdot \mathbf{L}^T = \mathbf{A} \tag{2.9.2}$$

This factorization is sometimes referred to as "taking the square root" of the matrix $\mathbf{A}$. The components of $\mathbf{L}^T$ are of course related to those of $\mathbf{L}$ by

$$L^T_{ij} = L_{ji} \tag{2.9.3}$$

Writing out equation (2.9.2) in components, one readily obtains the analogs of equations (2.3.12)–(2.3.13),

$$L_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \right)^{1/2} \tag{2.9.4}$$

and

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right) \qquad j = i+1, i+2, \ldots, N \tag{2.9.5}$$

If you apply equations (2.9.4) and (2.9.5) in the order $i = 1, 2, \ldots, N$, you will see that the L's that occur on the right-hand side are already determined by the time they are needed. Also, only components a_{ij} with $j \geq i$ are referenced. (Since $\mathbf{A}$ is symmetric, these have complete information.) It is convenient, then, to have the factor $\mathbf{L}$ overwrite the subdiagonal (lower triangular but not including the diagonal) part of $\mathbf{A}$, preserving the input upper triangular values of $\mathbf{A}$. Only one extra vector of length N is needed to store the diagonal part of $\mathbf{L}$. The operations count is $N^3/6$ executions of the inner loop (consisting of one multiply and one subtract), with also N square roots. As already mentioned, this is about a factor 2 better than LU decomposition of $\mathbf{A}$ (where its symmetry would be ignored).

A straightforward implementation is

```
SUBROUTINE choldc(a,n,np,p)
INTEGER n,np
REAL a(np,np),p(n)
    Given a positive-definite symmetric matrix a(1:n,1:n), with physical dimension np, this
    routine constructs its Cholesky decomposition, A = L·L^T. On input, only the upper triangle
    of a need be given; it is not modified. The Cholesky factor L is returned in the lower triangle
    of a, except for its diagonal elements which are returned in p(1:n).
INTEGER i,j,k
REAL sum
do 13 i=1,n
    do 12 j=i,n
        sum=a(i,j)
        do 11 k=i-1,1,-1
            sum=sum-a(i,k)*a(j,k)
        enddo 11
        if(i.eq.j)then
            if(sum.le.0.)pause 'choldc failed'       a, with rounding errors, is not
            p(i)=sqrt(sum)                             positive definite.
        else
            a(j,i)=sum/p(i)
        endif
    enddo 12
enddo 13
return
END
```

You might at this point wonder about pivoting. The pleasant answer is that Cholesky decomposition is extremely stable numerically, without any pivoting at all. Failure of `choldc` simply indicates that the matrix $\mathbf{A}$ (or, with roundoff error, another very nearby matrix) is not positive definite. In fact, `choldc` is an efficient way to test *whether* a symmetric matrix is positive definite. (In this application, you will want to replace the `pause` with some less drastic signaling method.)

Once your matrix is decomposed, the triangular factor can be used to solve a linear equation by backsubstitution. The straightforward implementation of this is

```
SUBROUTINE cholsl(a,n,np,p,b,x)
INTEGER n,np
REAL a(np,np),b(n),p(n),x(n)
    Solves the set of n linear equations A · x = b, where a is a positive-definite symmetric
    matrix with physical dimension np. a and p are input as the output of the routine choldc.
    Only the lower triangle of a is accessed. b(1:n) is input as the right-hand side vector. The
    solution vector is returned in x(1:n). a, n, np, and p are not modified and can be left
    in place for successive calls with different right-hand sides b. b is not modified unless you
    identify b and x in the calling sequence, which is allowed.
INTEGER i,k
REAL sum
do 12 i=1,n                              Solve L · y = b, storing y in x.
    sum=b(i)
    do 11 k=i-1,1,-1
        sum=sum-a(i,k)*x(k)
```

```
    enddo 11
    x(i)=sum/p(i)
enddo 12
do 14 i=n,1,-1                    Solve L^T · x = y.
    sum=x(i)
    do 13 k=i+1,n
        sum=sum-a(k,i)*x(k)
    enddo 13
    x(i)=sum/p(i)
enddo 14
return
END
```

A typical use of `choldc` and `cholsl` is in the inversion of covariance matrices describing the fit of data to a model; see, e.g., §15.6. In this, and many other applications, one often needs $\mathbf{L}^{-1}$. The lower triangle of this matrix can be efficiently found from the output of `choldc`:

```
    do 13 i=1,n
        a(i,i)=1./p(i)
        do 12 j=i+1,n
            sum=0.
            do 11 k=i,j-1
                sum=sum-a(j,k)*a(k,i)
            enddo 11
            a(j,i)=sum/p(j)
        enddo 12
    enddo 13
```

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/1.

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), §4.9.2.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.5.

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.2.

2.10 QR Decomposition

There is another matrix factorization that is sometimes very useful, the so-called QR *decomposition*,

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{2.10.1}$$

Here $\mathbf{R}$ is upper triangular, while $\mathbf{Q}$ is orthogonal, that is,

$$\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{1} \tag{2.10.2}$$

where $\mathbf{Q}^T$ is the transpose matrix of $\mathbf{Q}$. Although the decomposition exists for a general rectangular matrix, we shall restrict our treatment to the case when all the matrices are square, with dimensions $N \times N$.

```
    enddo 11
    x(i)=sum/p(i)
enddo 12
do 14 i=n,1,-1                    Solve L^T · x = y.
    sum=x(i)
    do 13 k=i+1,n
        sum=sum-a(k,i)*x(k)
    enddo 13
    x(i)=sum/p(i)
enddo 14
return
END
```

A typical use of `choldc` and `cholsl` is in the inversion of covariance matrices describing the fit of data to a model; see, e.g., §15.6. In this, and many other applications, one often needs $\mathbf{L}^{-1}$. The lower triangle of this matrix can be efficiently found from the output of `choldc`:

```
    do 13 i=1,n
        a(i,i)=1./p(i)
        do 12 j=i+1,n
            sum=0.
            do 11 k=i,j-1
                sum=sum-a(j,k)*a(k,i)
            enddo 11
            a(j,i)=sum/p(j)
        enddo 12
    enddo 13
```

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/1.

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), §4.9.2.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.5.

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.2.

2.10 QR Decomposition

There is another matrix factorization that is sometimes very useful, the so-called QR *decomposition*,

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{2.10.1}$$

Here $\mathbf{R}$ is upper triangular, while $\mathbf{Q}$ is orthogonal, that is,

$$\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{1} \tag{2.10.2}$$

where $\mathbf{Q}^T$ is the transpose matrix of $\mathbf{Q}$. Although the decomposition exists for a general rectangular matrix, we shall restrict our treatment to the case when all the matrices are square, with dimensions $N \times N$.

Like the other matrix factorizations we have met (LU, SVD, Cholesky), QR decomposition can be used to solve systems of linear equations. To solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.10.3}$$

first form $\mathbf{Q}^T \cdot \mathbf{b}$ and then solve

$$\mathbf{R} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b} \tag{2.10.4}$$

by backsubstitution. Since QR decomposition involves about twice as many operations as LU decomposition, it is not used for typical systems of linear equations. However, we will meet special cases where QR is the method of choice.

The standard algorithm for the QR decomposition involves successive Householder transformations (to be discussed later in §11.2). We write a Householder matrix in the form $\mathbf{1} - \mathbf{u} \otimes \mathbf{u}/c$ where $c = \frac{1}{2}\mathbf{u} \cdot \mathbf{u}$. An appropriate Householder matrix applied to a given matrix can zero all elements in a column of the matrix situated below a chosen element. Thus we arrange for the first Householder matrix $\mathbf{Q}_1$ to zero all elements in the first column of $\mathbf{A}$ below the first element. Similarly $\mathbf{Q}_2$ zeroes all elements in the second column below the second element, and so on up to $\mathbf{Q}_{n-1}$. Thus

$$\mathbf{R} = \mathbf{Q}_{n-1} \cdots \mathbf{Q}_1 \cdot \mathbf{A} \tag{2.10.5}$$

Since the Householder matrices are orthogonal,

$$\mathbf{Q} = (\mathbf{Q}_{n-1} \cdots \mathbf{Q}_1)^{-1} = \mathbf{Q}_1 \cdots \mathbf{Q}_{n-1} \tag{2.10.6}$$

In most applications we don't need to form $\mathbf{Q}$ explicitly; we instead store it in the factored form (2.10.6). Pivoting is not usually necessary unless the matrix $\mathbf{A}$ is very close to singular. A general QR algorithm for rectangular matrices including pivoting is given in [1]. For square matrices, an implementation is the following:

```
SUBROUTINE qrdcmp(a,n,np,c,d,sing)
INTEGER n,np
REAL a(np,np),c(n),d(n)
LOGICAL sing
```

Constructs the QR decomposition of `a(1:n,1:n)`, with physical dimension `np`. The upper triangular matrix $\mathbf{R}$ is returned in the upper triangle of `a`, except for the diagonal elements of $\mathbf{R}$ which are returned in `d(1:n)`. The orthogonal matrix $\mathbf{Q}$ is represented as a product of $n-1$ Householder matrices $\mathbf{Q}_1 \ldots \mathbf{Q}_{n-1}$, where $\mathbf{Q}_j = \mathbf{1} - \mathbf{u}_j \otimes \mathbf{u}_j / c_j$. The ith component of $\mathbf{u}_j$ is zero for $i = 1, \ldots, j-1$ while the nonzero components are returned in `a(i,j)` for $i = j, \ldots, n$. `sing` returns as true if singularity is encountered during the decomposition, but the decomposition is still completed in this case.

```
INTEGER i,j,k
REAL scale,sigma,sum,tau
sing=.false.
do 17 k=1,n-1
    scale=0.
    do 11 i=k,n
        scale=max(scale,abs(a(i,k)))
    enddo 11
    if(scale.eq.0.)then                 Singular case.
        sing=.true.
        c(k)=0.
        d(k)=0.
    else                                Form Q_k and Q_k · A.
        do 12 i=k,n
            a(i,k)=a(i,k)/scale
        enddo 12
        sum=0.
        do 13 i=k,n
            sum=sum+a(i,k)**2
        enddo 13
        sigma=sign(sqrt(sum),a(k,k))
        a(k,k)=a(k,k)+sigma
```

```
            c(k)=sigma*a(k,k)
            d(k)=-scale*sigma
            do 16 j=k+1,n
                sum=0.
                do 14 i=k,n
                    sum=sum+a(i,k)*a(i,j)
                enddo 14
                tau=sum/c(k)
                do 15 i=k,n
                    a(i,j)=a(i,j)-tau*a(i,k)
                enddo 15
            enddo 16
        endif
    enddo 17
    d(n)=a(n,n)
    if(d(n).eq.0.)sing=.true.
    return
    END
```

The next routine, `qrsolv`, is used to solve linear systems. In many applications only the part (2.10.4) of the algorithm is needed, so we separate it off into its own routine `rsolv`.

```
    SUBROUTINE qrsolv(a,n,np,c,d,b)
    INTEGER n,np
    REAL a(np,np),b(n),c(n),d(n)
C   USES rsolv
```

Solves the set of `n` linear equations $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$, where `a` is a matrix with physical dimension `np`. `a`, `c`, and `d` are input as the output of the routine `qrdcmp` and are not modified. `b(1:n)` is input as the right-hand side vector, and is overwritten with the solution vector on output.

```
    INTEGER i,j
    REAL sum,tau
    do 13 j=1,n-1                       Form Q^T · b.
        sum=0.
        do 11 i=j,n
            sum=sum+a(i,j)*b(i)
        enddo 11
        tau=sum/c(j)
        do 12 i=j,n
            b(i)=b(i)-tau*a(i,j)
        enddo 12
    enddo 13
    call rsolv(a,n,np,d,b)              Solve R · x = Q^T · b.
    return
    END
```

```
    SUBROUTINE rsolv(a,n,np,d,b)
    INTEGER n,np
    REAL a(np,np),b(n),d(n)
```

Solves the set of `n` linear equations $\mathbf{R}\cdot\mathbf{x}=\mathbf{b}$, where $\mathbf{R}$ is an upper triangular matrix stored in `a` and `d`. `a` and `d` are input as the output of the routine `qrdcmp` and are not modified. `b(1:n)` is input as the right-hand side vector, and is overwritten with the solution vector on output.

```
    INTEGER i,j
    REAL sum
    b(n)=b(n)/d(n)
    do 12 i=n-1,1,-1
        sum=0.
        do 11 j=i+1,n
            sum=sum+a(i,j)*b(j)
        enddo 11
        b(i)=(b(i)-sum)/d(i)
```

```
      enddo 12
      return
      END
```

See [2] for details on how to use QR decomposition for constructing orthogonal bases, and for solving least-squares problems. (We prefer to use SVD, §2.6, for these purposes, because of its greater diagnostic capability in pathological cases.)

Updating a QR decomposition

Some numerical algorithms involve solving a succession of linear systems each of which differs only slightly from its predecessor. Instead of doing $O(N^3)$ operations each time to solve the equations from scratch, one can often update a matrix factorization in $O(N^2)$ operations and use the new factorization to solve the next set of linear equations. The LU decomposition is complicated to update because of pivoting. However, QR turns out to be quite simple for a very common kind of update,

$$\mathbf{A} \to \mathbf{A} + \mathbf{s} \otimes \mathbf{t} \tag{2.10.7}$$

(compare equation 2.7.1). In practice it is more convenient to work with the equivalent form

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad \to \quad \mathbf{A}' = \mathbf{Q}' \cdot \mathbf{R}' = \mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v}) \tag{2.10.8}$$

One can go back and forth between equations (2.10.7) and (2.10.8) using the fact that $\mathbf{Q}$ is orthogonal, giving

$$\mathbf{t} = \mathbf{v} \quad \text{and either} \quad \mathbf{s} = \mathbf{Q} \cdot \mathbf{u} \quad \text{or} \quad \mathbf{u} = \mathbf{Q}^T \cdot \mathbf{s} \tag{2.10.9}$$

The algorithm [2] has two phases. In the first we apply $N-1$ Jacobi rotations (§11.1) to reduce $\mathbf{R} + \mathbf{u} \otimes \mathbf{v}$ to upper Hessenberg form. Another $N-1$ Jacobi rotations transform this upper Hessenberg matrix to the new upper triangular matrix $\mathbf{R}'$. The matrix $\mathbf{Q}'$ is simply the product of $\mathbf{Q}$ with the $2(N-1)$ Jacobi rotations. In applications we usually want $\mathbf{Q}^T$, and the algorithm can easily be rearranged to work with this matrix instead of with $\mathbf{Q}$.

```
      SUBROUTINE qrupdt(r,qt,n,np,u,v)
      INTEGER n,np
      REAL r(np,np),qt(np,np),u(np),v(np)
C     USES rotate
```

Given the QR decomposition of some n × n matrix, calculates the QR decomposition of the matrix $\mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v})$. The matrices r and qt have physical dimension np. Note that $\mathbf{Q}^T$ is input and returned in qt.

```
      INTEGER i,j,k
      do 11 k=n,1,-1                                  Find largest k such that u(k) ≠ 0.
        if(u(k).ne.0.)goto 1
      enddo 11
      k=1
1     do 12 i=k-1,1,-1                                Transform R + u ⊗ v to upper Hes-
        call rotate(r,qt,n,np,i,u(i),-u(i+1))           senberg.
        if(u(i).eq.0.)then
          u(i)=abs(u(i+1))
        else if(abs(u(i)).gt.abs(u(i+1)))then
          u(i)=abs(u(i))*sqrt(1.+(u(i+1)/u(i))**2)
        else
          u(i)=abs(u(i+1))*sqrt(1.+(u(i)/u(i+1))**2)
        endif
      enddo 12
      do 13 j=1,n
        r(1,j)=r(1,j)+u(1)*v(j)
      enddo 13
      do 14 i=1,k-1                                   Transform upper Hessenberg matrix
        call rotate(r,qt,n,np,i,r(i,i),-r(i+1,i))       to upper triangular.
      enddo 14
```

```
return
END



SUBROUTINE rotate(r,qt,n,np,i,a,b)
INTEGER n,np,i
REAL a,b,r(np,np),qt(np,np)
    Given n×n matrices r and qt of physical dimension np, carry out a Jacobi rotation on rows i
    and i+1 of each matrix. a and b are the parameters of the rotation: cos θ = a/√(a²+b²),
    sin θ = b/√(a²+b²).
INTEGER j
REAL c,fact,s,w,y
if(a.eq.0.)then                          Avoid unnecessary overflow or underflow.
    c=0.
    s=sign(1.,b)
else if(abs(a).gt.abs(b))then
    fact=b/a
    c=sign(1./sqrt(1.+fact**2),a)
    s=fact*c
else
    fact=a/b
    s=sign(1./sqrt(1.+fact**2),b)
    c=fact*s
endif
do 11 j=i,n                              Premultiply r by Jacobi rotation.
    y=r(i,j)
    w=r(i+1,j)
    r(i,j)=c*y-s*w
    r(i+1,j)=s*y+c*w
enddo 11
do 12 j=1,n                              Premultiply qt by Jacobi rotation.
    y=qt(i,j)
    w=qt(i+1,j)
    qt(i,j)=c*y-s*w
    qt(i+1,j)=s*y+c*w
enddo 12
return
END
```

We will make use of QR decomposition, and its updating, in §9.7.

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/8. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §§5.2, 5.3, 12.6. [2]

2.11 Is Matrix Inversion an N^3 Process?

We close this chapter with a little entertainment, a bit of algorithmic prestidigitation which probes more deeply into the subject of matrix inversion. We start with a seemingly simple question:

```
return
END



SUBROUTINE rotate(r,qt,n,np,i,a,b)
INTEGER n,np,i
REAL a,b,r(np,np),qt(np,np)
    Given n×n matrices r and qt of physical dimension np, carry out a Jacobi rotation on rows i
    and i+1 of each matrix. a and b are the parameters of the rotation: cos θ = a/√(a²+b²),
    sin θ = b/√(a²+b²).
INTEGER j
REAL c,fact,s,w,y
if(a.eq.0.)then                          Avoid unnecessary overflow or underflow.
    c=0.
    s=sign(1.,b)
else if(abs(a).gt.abs(b))then
    fact=b/a
    c=sign(1./sqrt(1.+fact**2),a)
    s=fact*c
else
    fact=a/b
    s=sign(1./sqrt(1.+fact**2),b)
    c=fact*s
endif
do 11 j=i,n                              Premultiply r by Jacobi rotation.
    y=r(i,j)
    w=r(i+1,j)
    r(i,j)=c*y-s*w
    r(i+1,j)=s*y+c*w
enddo 11
do 12 j=1,n                              Premultiply qt by Jacobi rotation.
    y=qt(i,j)
    w=qt(i+1,j)
    qt(i,j)=c*y-s*w
    qt(i+1,j)=s*y+c*w
enddo 12
return
END
```

We will make use of QR decomposition, and its updating, in §9.7.

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/8. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §§5.2, 5.3, 12.6. [2]

2.11 Is Matrix Inversion an N^3 Process?

We close this chapter with a little entertainment, a bit of algorithmic prestidigitation which probes more deeply into the subject of matrix inversion. We start with a seemingly simple question:

How many individual multiplications does it take to perform the matrix multiplication of two 2×2 matrices,

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \tag{2.11.1}$$

Eight, right? Here they are written explicitly:

$$\begin{aligned} c_{11} &= a_{11} \times b_{11} + a_{12} \times b_{21} \\ c_{12} &= a_{11} \times b_{12} + a_{12} \times b_{22} \\ c_{21} &= a_{21} \times b_{11} + a_{22} \times b_{21} \\ c_{22} &= a_{21} \times b_{12} + a_{22} \times b_{22} \end{aligned} \tag{2.11.2}$$

Do you think that one can write formulas for the c's that involve only *seven* multiplications? (Try it yourself, before reading on.)

Such a set of formulas was, in fact, discovered by Strassen [1]. The formulas are:

$$\begin{aligned} Q_1 &\equiv (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\ Q_2 &\equiv (a_{21} + a_{22}) \times b_{11} \\ Q_3 &\equiv a_{11} \times (b_{12} - b_{22}) \\ Q_4 &\equiv a_{22} \times (-b_{11} + b_{21}) \\ Q_5 &\equiv (a_{11} + a_{12}) \times b_{22} \\ Q_6 &\equiv (-a_{11} + a_{21}) \times (b_{11} + b_{12}) \\ Q_7 &\equiv (a_{12} - a_{22}) \times (b_{21} + b_{22}) \end{aligned} \tag{2.11.3}$$

in terms of which

$$\begin{aligned} c_{11} &= Q_1 + Q_4 - Q_5 + Q_7 \\ c_{21} &= Q_2 + Q_4 \\ c_{12} &= Q_3 + Q_5 \\ c_{22} &= Q_1 + Q_3 - Q_2 + Q_6 \end{aligned} \tag{2.11.4}$$

What's the use of this? There is one fewer multiplication than in equation (2.11.2), but *many more* additions and subtractions. It is not clear that anything has been gained. But notice that in (2.11.3) the a's and b's are never commuted. Therefore (2.11.3) and (2.11.4) are valid when the a's and b's are themselves matrices. The problem of multiplying two very large matrices (of order $N = 2^m$ for some integer *m*) can now be broken down recursively by partitioning the matrices into quarters, sixteenths, etc. And note the key point: The savings is not just a factor "7/8"; it is that factor at *each* hierarchical level of the recursion. In total it reduces the process of matrix multiplication to order $N^{\log_2 7}$ instead of N^3.

What about all the extra additions in (2.11.3)–(2.11.4)? Don't they outweigh the advantage of the fewer multiplications? For large N, it turns out that there are six times as many additions as multiplications implied by (2.11.3)–(2.11.4). But, if N is very large, this constant factor is no match for the change in the *exponent* from N^3 to $N^{\log_2 7}$.

With this "fast" matrix multiplication, Strassen also obtained a surprising result for matrix inversion [1]. Suppose that the matrices

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \tag{2.11.5}$$

are inverses of each other. Then the c's can be obtained from the a's by the following operations (compare equations 2.7.22 and 2.7.25):

$$\begin{aligned} R_1 &= \text{Inverse}(a_{11}) \\ R_2 &= a_{21} \times R_1 \\ R_3 &= R_1 \times a_{12} \\ R_4 &= a_{21} \times R_3 \\ R_5 &= R_4 - a_{22} \\ R_6 &= \text{Inverse}(R_5) \\ c_{12} &= R_3 \times R_6 \\ c_{21} &= R_6 \times R_2 \\ R_7 &= R_3 \times c_{21} \\ c_{11} &= R_1 - R_7 \\ c_{22} &= -R_6 \end{aligned} \tag{2.11.6}$$

In (2.11.6) the "inverse" operator occurs just twice. It is to be interpreted as the reciprocal if the a's and c's are scalars, but as matrix inversion if the a's and c's are themselves submatrices. Imagine doing the inversion of a very large matrix, of order $N = 2^m$, recursively by partitions in half. At each step, halving the order *doubles* the number of inverse operations. But this means that there are only N divisions in all! So divisions don't dominate in the recursive use of (2.11.6). Equation (2.11.6) is dominated, in fact, by its 6 multiplications. Since these can be done by an $N^{\log_2 7}$ algorithm, so can the matrix inversion!

This is fun, but let's look at practicalities: If you estimate how large N has to be before the difference between exponent 3 and exponent $\log_2 7 = 2.807$ is substantial enough to outweigh the bookkeeping overhead, arising from the complicated nature of the recursive Strassen algorithm, you will find that LU decomposition is in no immediate danger of becoming obsolete.

If, on the other hand, you like this kind of fun, then try these: (1) Can you multiply the complex numbers $(a+ib)$ and $(c+id)$ in only *three* real multiplications? [Answer: see §5.4.] (2) Can you evaluate a general fourth-degree polynomial in

x for many different values of x with only *three* multiplications per evaluation? [Answer: see §5.3.]

CITED REFERENCES AND FURTHER READING:

Strassen, V. 1969, *Numerische Mathematik*, vol. 13, pp. 354–356. [1]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).

Winograd, S. 1971, *Linear Algebra and Its Applications*, vol. 4, pp. 381–388.

Pan, V. Ya. 1980, *SIAM Journal on Computing*, vol. 9, pp. 321–342.

Pan, V. 1984, *How to Multiply Matrices Faster*, Lecture Notes in Computer Science, vol. 179 (New York: Springer-Verlag)

Pan, V. 1984, *SIAM Review*, vol. 26, pp. 393–415. [More recent results that show that an exponent of 2.496 can be achieved — theoretically!]

Chapter 3. Interpolation and Extrapolation

3.0 Introduction

We sometimes know the value of a function $f(x)$ at a set of points $x_1, x_2, \ldots, x_N$ (say, with $x_1 < \ldots < x_N$), but we don't have an analytic expression for $f(x)$ that lets us calculate its value at an arbitrary point. For example, the $f(x_i)$'s might result from some physical measurement or from long numerical calculation that cannot be cast into a simple functional form. Often the x_i's are equally spaced, but not necessarily.

The task now is to estimate $f(x)$ for arbitrary x by, in some sense, drawing a smooth curve through (and perhaps beyond) the x_i. If the desired x is in between the largest and smallest of the x_i's, the problem is called *interpolation*; if x is outside that range, it is called *extrapolation*, which is considerably more hazardous (as many former stock-market analysts can attest).

Interpolation and extrapolation schemes must model the function, between or beyond the known points, by some plausible functional form. The form should be sufficiently general so as to be able to approximate large classes of functions which might arise in practice. By far most common among the functional forms used are polynomials (§3.1). Rational functions (quotients of polynomials) also turn out to be extremely useful (§3.2). Trigonometric functions, sines and cosines, give rise to *trigonometric interpolation* and related Fourier methods, which we defer to Chapters 12 and 13.

There is an extensive mathematical literature devoted to theorems about what sort of functions can be well approximated by which interpolating functions. These theorems are, alas, almost completely useless in day-to-day work: If we know enough about our function to apply a theorem of any power, we are usually not in the pitiful state of having to interpolate on a table of its values!

Interpolation is related to, but distinct from, *function approximation*. That task consists of finding an approximate (but easily computable) function to use in place of a more complicated one. In the case of interpolation, you are given the function f at points *not of your own choosing*. For the case of function approximation, you are allowed to compute the function f at *any* desired points for the purpose of developing your approximation. We deal with function approximation in Chapter 5.

One can easily find pathological functions that make a mockery of any interpolation scheme. Consider, for example, the function

$$f(x) = 3x^2 + \frac{1}{\pi^4} \ln\left[(\pi - x)^2\right] + 1 \tag{3.0.1}$$

which is well-behaved everywhere except at $x = \pi$, very mildly singular at $x = \pi$, and otherwise takes on all positive and negative values. Any interpolation based on the values $x = 3.13, 3.14, 3.15, 3.16$, will assuredly get a very wrong answer for the value $x = 3.1416$, even though a graph plotting those five points looks really quite smooth! (Try it on your calculator.)

Because pathologies can lurk anywhere, it is highly desirable that an interpolation and extrapolation routine should return an estimate of its own error. Such an error estimate can never be foolproof, of course. We could have a function that, for reasons known only to its maker, takes off wildly and unexpectedly between two tabulated points. Interpolation always presumes some degree of smoothness for the function interpolated, but within this framework of presumption, deviations from smoothness can be detected.

Conceptually, the interpolation process has two stages: (1) Fit an interpolating function to the data points provided. (2) Evaluate that interpolating function at the target point x.

However, this two-stage method is generally not the best way to proceed in practice. Typically it is computationally less efficient, and more susceptible to roundoff error, than methods which construct a functional estimate $f(x)$ directly from the N tabulated values every time one is desired. Most practical schemes start at a nearby point $f(x_i)$, then add a sequence of (hopefully) decreasing corrections, as information from other $f(x_i)$'s is incorporated. The procedure typically takes $O(N^2)$ operations. If everything is well behaved, the last correction will be the smallest, and it can be used as an informal (though not rigorous) bound on the error.

In the case of polynomial interpolation, it sometimes does happen that the coefficients of the interpolating polynomial are of interest, even though their use in *evaluating* the interpolating function should be frowned on. We deal with this eventuality in §3.5.

Local interpolation, using a finite number of "nearest-neighbor" points, gives interpolated values $f(x)$ that do not, in general, have continuous first or higher derivatives. That happens because, as x crosses the tabulated values x_i, the interpolation scheme switches which tabulated points are the "local" ones. (If such a switch is allowed to occur anywhere *else*, then there will be a discontinuity in the interpolated function itself at that point. Bad idea!)

In situations where continuity of derivatives is a concern, one must use the "stiffer" interpolation provided by a so-called *spline* function. A spline is a polynomial between each pair of table points, but one whose coefficients are determined "slightly" nonlocally. The nonlocality is designed to guarantee global smoothness in the interpolated function up to some order of derivative. Cubic splines (§3.3) are the most popular. They produce an interpolated function that is continuous through the second derivative. Splines tend to be stabler than polynomials, with less possibility of wild oscillation between the tabulated points.

The number of points (minus one) used in an interpolation scheme is called the *order* of the interpolation. Increasing the order does not necessarily increase the accuracy, especially in polynomial interpolation. If the added points are distant from the point of interest x, the resulting higher-order polynomial, with its additional constrained points, tends to oscillate wildly between the tabulated values. This oscillation may have no relation at all to the behavior of the "true" function (see Figure 3.0.1). Of course, adding points *close* to the desired point usually does help,

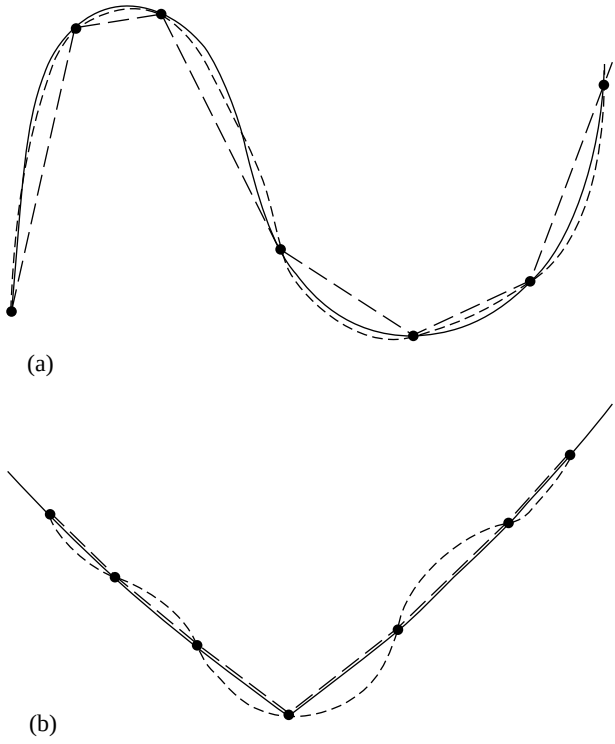


Figure 3.0.1. (a) A smooth function (solid line) is more accurately interpolated by a high-order polynomial (shown schematically as dotted line) than by a low-order polynomial (shown as a piecewise linear dashed line). (b) A function with sharp corners or rapidly changing higher derivatives is *less* accurately approximated by a high-order polynomial (dotted line), which is too "stiff," than by a low-order polynomial (dashed lines). Even some smooth functions, such as exponentials or rational functions, can be badly approximated by high-order polynomials.

but a finer mesh implies a larger table of values, not always available.

Unless there is solid evidence that the interpolating function is close in form to the true function f, it is a good idea to be cautious about high-order interpolation. We enthusiastically endorse interpolations with 3 or 4 points, we are perhaps tolerant of 5 or 6; but we rarely go higher than that unless there is quite rigorous monitoring of estimated errors.

When your table of values contains many more points than the desirable order of interpolation, you must begin each interpolation with a search for the right "local" place in the table. While not strictly a part of the subject of interpolation, this task is important enough (and often enough botched) that we devote §3.4 to its discussion.

The routines given for interpolation are also routines for extrapolation. An important application, in Chapter 16, is their use in the integration of ordinary differential equations. There, considerable care *is* taken with the monitoring of errors. Otherwise, the dangers of extrapolation cannot be overemphasized: An interpolating function, which is perforce an extrapolating function, will typically go berserk when the argument x is outside the range of tabulated values by more than the typical spacing of tabulated points.

Interpolation can be done in more than one dimension, e.g., for a function

$f(x, y, z)$. Multidimensional interpolation is often accomplished by a sequence of one-dimensional interpolations. We discuss this in §3.6.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 2.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 3.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 4.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 5.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 3.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 6.

3.1 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points, a unique quadratic. Et cetera. The interpolating polynomial of degree $N-1$ through the N points $y_1 = f(x_1), y_2 = f(x_2), \ldots, y_N = f(x_N)$ is given explicitly by Lagrange's classical formula,

$$P(x) = \frac{(x-x_2)(x-x_3)\ldots(x-x_N)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_N)}y_1 + \frac{(x-x_1)(x-x_3)\ldots(x-x_N)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_N)}y_2 + \cdots + \frac{(x-x_1)(x-x_2)\ldots(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)\ldots(x_N-x_{N-1})}y_N \tag{3.1.1}$$

There are N terms, each a polynomial of degree $N-1$ and each constructed to be zero at all of the x_i except one, at which it is constructed to be y_i.

It is not terribly wrong to implement the Lagrange formula straightforwardly, but it is not terribly right either. The resulting algorithm gives no error estimate, and it is also somewhat awkward to program. A much better algorithm (for constructing the same, unique, interpolating polynomial) is *Neville's algorithm*, closely related to and sometimes confused with *Aitken's algorithm*, the latter now considered obsolete.

Let P_1 be the value at x of the unique polynomial of degree zero (i.e., a constant) passing through the point (x_1, y_1); so $P_1 = y_1$. Likewise define $P_2, P_3, \ldots, P_N$. Now let P_{12} be the value at x of the unique polynomial of degree one passing through both (x_1, y_1) and (x_2, y_2). Likewise $P_{23}, P_{34}, \ldots, P_{(N-1)N}$. Similarly, for higher-order polynomials, up to $P_{123\ldots N}$, which is the value of the unique interpolating polynomial through all N points, i.e., the desired answer.

$f(x, y, z)$. Multidimensional interpolation is often accomplished by a sequence of one-dimensional interpolations. We discuss this in §3.6.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 2.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 3.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 4.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 5.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 3.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 6.

3.1 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points, a unique quadratic. Et cetera. The interpolating polynomial of degree $N-1$ through the N points $y_1 = f(x_1), y_2 = f(x_2), \ldots, y_N = f(x_N)$ is given explicitly by Lagrange's classical formula,

$$P(x) = \frac{(x-x_2)(x-x_3)\ldots(x-x_N)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_N)}y_1 + \frac{(x-x_1)(x-x_3)\ldots(x-x_N)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_N)}y_2 + \cdots + \frac{(x-x_1)(x-x_2)\ldots(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)\ldots(x_N-x_{N-1})}y_N \tag{3.1.1}$$

There are N terms, each a polynomial of degree $N-1$ and each constructed to be zero at all of the x_i except one, at which it is constructed to be y_i.

It is not terribly wrong to implement the Lagrange formula straightforwardly, but it is not terribly right either. The resulting algorithm gives no error estimate, and it is also somewhat awkward to program. A much better algorithm (for constructing the same, unique, interpolating polynomial) is *Neville's algorithm*, closely related to and sometimes confused with *Aitken's algorithm*, the latter now considered obsolete.

Let P_1 be the value at x of the unique polynomial of degree zero (i.e., a constant) passing through the point (x_1, y_1); so $P_1 = y_1$. Likewise define $P_2, P_3, \ldots, P_N$. Now let P_{12} be the value at x of the unique polynomial of degree one passing through both (x_1, y_1) and (x_2, y_2). Likewise $P_{23}, P_{34}, \ldots, P_{(N-1)N}$. Similarly, for higher-order polynomials, up to $P_{123\ldots N}$, which is the value of the unique interpolating polynomial through all N points, i.e., the desired answer.

The various P's form a "tableau" with "ancestors" on the left leading to a single "descendant" at the extreme right. For example, with $N = 4$,

$$\begin{array}{lllll} x_1: & y_1 = P_1 & & & \\ & & P_{12} & & \\ x_2: & y_2 = P_2 & & P_{123} & \\ & & P_{23} & & P_{1234} \\ x_3: & y_3 = P_3 & & P_{234} & \\ & & P_{34} & & \\ x_4: & y_4 = P_4 & & & \end{array} \tag{3.1.2}$$

Neville's algorithm is a recursive way of filling in the numbers in the tableau a column at a time, from left to right. It is based on the relationship between a "daughter" P and its two "parents,"

$$P_{i(i+1)\ldots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\ldots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\ldots(i+m)}}{x_i - x_{i+m}} \tag{3.1.3}$$

This recurrence works because the two parents already agree at points $x_{i+1} \ldots x_{i+m-1}$.

An improvement on the recurrence (3.1.3) is to keep track of the small *differences* between parents and daughters, namely to define (for $m = 1, 2, \ldots, N - 1$),

$$\begin{aligned} C_{m,i} &\equiv P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)} \\ D_{m,i} &\equiv P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)}. \end{aligned} \tag{3.1.4}$$

Then one can easily derive from (3.1.3) the relations

$$\begin{aligned} D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\ C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \end{aligned} \tag{3.1.5}$$

At each level m, the C's and D's are the corrections that make the interpolation one order higher. The final answer $P_{1\ldots N}$ is equal to the sum of *any* y_i plus a set of C's and/or D's that form a path through the family tree to the rightmost daughter.

Here is a routine for polynomial interpolation or extrapolation:

```
SUBROUTINE polint(xa,ya,n,x,y,dy)
INTEGER n,NMAX
REAL dy,x,y,xa(n),ya(n)
PARAMETER (NMAX=10)              Largest anticipated value of n.
    Given arrays xa and ya, each of length n, and given a value x, this routine returns a
    value y, and an error estimate dy. If P(x) is the polynomial of degree N − 1 such that
    P(xa_i) = ya_i, i = 1, ..., n, then the returned value y = P(x).
INTEGER i,m,ns
REAL den,dif,dift,ho,hp,w,c(NMAX),d(NMAX)
ns=1
dif=abs(x-xa(1))
```

```
do 11 i=1,n                    Here we find the index ns of the closest table entry,
    dift=abs(x-xa(i))
    if (dift.lt.dif) then
        ns=i
        dif=dift
    endif
    c(i)=ya(i)                 and initialize the tableau of c's and d's.
    d(i)=ya(i)
enddo 11
y=ya(ns)                       This is the initial approximation to y.
ns=ns-1
do 13 m=1,n-1                  For each column of the tableau,
    do 12 i=1,n-m              we loop over the current c's and d's and update them.
        ho=xa(i)-x
        hp=xa(i+m)-x
        w=c(i+1)-d(i)
        den=ho-hp
        if(den.eq.0.)pause 'failure in polint'
          This error can occur only if two input xa's are (to within roundoff) identical.
        den=w/den
        d(i)=hp*den            Here the c's and d's are updated.
        c(i)=ho*den
    enddo 12
    if (2*ns.lt.n-m)then       After each column in the tableau is completed, we decide
        dy=c(ns+1)                 which correction, c or d, we want to add to our accu-
    else                           mulating value of y, i.e., which path to take through
        dy=d(ns)                   the tableau—forking up or down. We do this in such a
        ns=ns-1                    way as to take the most "straight line" route through the
    endif                          tableau to its apex, updating ns accordingly to keep track
    y=y+dy                         of where we are. This route keeps the partial approxima-
enddo 13                           tions centered (insofar as possible) on the target x. The
return                             last dy added is thus the error indication.
END
```

Quite often you will want to call `polint` with the dummy arguments `xa` and `ya` replaced by actual arrays *with offsets*. For example, the construction `call polint(xx(15),yy(15),4,x,y,dy)` performs 4-point interpolation on the tabulated values `xx(15:18)`, `yy(15:18)`. For more on this, see the end of §3.4.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.1.

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.

3.2 Rational Function Interpolation and Extrapolation

Some functions are not well approximated by polynomials, but *are* well approximated by rational functions, that is quotients of polynomials. We denote by $R_{i(i+1)\ldots(i+m)}$ a rational function passing through the $m+1$ points

```
do 11 i=1,n                    Here we find the index ns of the closest table entry,
    dift=abs(x-xa(i))
    if (dift.lt.dif) then
        ns=i
        dif=dift
    endif
    c(i)=ya(i)                 and initialize the tableau of c's and d's.
    d(i)=ya(i)
enddo 11
y=ya(ns)                       This is the initial approximation to y.
ns=ns-1
do 13 m=1,n-1                  For each column of the tableau,
    do 12 i=1,n-m              we loop over the current c's and d's and update them.
        ho=xa(i)-x
        hp=xa(i+m)-x
        w=c(i+1)-d(i)
        den=ho-hp
        if(den.eq.0.)pause 'failure in polint'
          This error can occur only if two input xa's are (to within roundoff) identical.
        den=w/den
        d(i)=hp*den            Here the c's and d's are updated.
        c(i)=ho*den
    enddo 12
    if (2*ns.lt.n-m)then       After each column in the tableau is completed, we decide
        dy=c(ns+1)                 which correction, c or d, we want to add to our accu-
    else                           mulating value of y, i.e., which path to take through
        dy=d(ns)                   the tableau—forking up or down. We do this in such a
        ns=ns-1                    way as to take the most "straight line" route through the
    endif                          tableau to its apex, updating ns accordingly to keep track
    y=y+dy                         of where we are. This route keeps the partial approxima-
enddo 13                           tions centered (insofar as possible) on the target x. The
return                             last dy added is thus the error indication.
END
```

Quite often you will want to call `polint` with the dummy arguments `xa` and `ya` replaced by actual arrays *with offsets*. For example, the construction `call polint(xx(15),yy(15),4,x,y,dy)` performs 4-point interpolation on the tabulated values `xx(15:18)`, `yy(15:18)`. For more on this, see the end of §3.4.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.1.

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.

3.2 Rational Function Interpolation and Extrapolation

Some functions are not well approximated by polynomials, but *are* well approximated by rational functions, that is quotients of polynomials. We denote by $R_{i(i+1)\ldots(i+m)}$ a rational function passing through the $m+1$ points

$(x_i, y_i) \ldots (x_{i+m}, y_{i+m})$. More explicitly, suppose

$$R_{i(i+1)\ldots(i+m)} = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \tag{3.2.1}$$

Since there are $\mu + \nu + 1$ unknown p's and q's (q_0 being arbitrary), we must have

$$m + 1 = \mu + \nu + 1 \tag{3.2.2}$$

In specifying a rational function interpolating function, you must give the desired order of both the numerator and the denominator.

Rational functions are sometimes superior to polynomials, roughly speaking, because of their ability to model functions with poles, that is, zeros of the denominator of equation (3.2.1). These poles might occur for real values of x, if the function to be interpolated itself has poles. More often, the function $f(x)$ is finite for all finite *real* x, but has an analytic continuation with poles in the complex x-plane. Such poles can themselves ruin a polynomial approximation, even one restricted to real values of x, just as they can ruin the convergence of an infinite power series in x. If you draw a circle in the complex plane around your m tabulated points, then you should not expect polynomial interpolation to be good unless the nearest pole is rather far outside the circle. A rational function approximation, by contrast, will stay "good" as long as it has enough powers of x in its denominator to account for (cancel) any nearby poles.

For the interpolation problem, a rational function is constructed so as to go through a chosen set of tabulated functional values. However, we should also mention in passing that rational function approximations can be used in analytic work. One sometimes constructs a rational function approximation by the criterion that the rational function of equation (3.2.1) itself have a power series expansion that agrees with the first $m + 1$ terms of the power series expansion of the desired function $f(x)$. This is called $Padé$ *approximation*, and is discussed in §5.12.

Bulirsch and Stoer found an algorithm of the Neville type which performs rational function extrapolation on tabulated data. A tableau like that of equation (3.1.2) is constructed column by column, leading to a result and an error estimate. The Bulirsch-Stoer algorithm produces the so-called *diagonal* rational function, with the degrees of numerator and denominator equal (if m is even) or with the degree of the denominator larger by one (if m is odd, cf. equation 3.2.2 above). For the derivation of the algorithm, refer to [1]. The algorithm is summarized by a recurrence relation exactly analogous to equation (3.1.3) for polynomial approximation:

$$R_{i(i+1)\ldots(i+m)} = R_{(i+1)\ldots(i+m)} + \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{\left(\frac{x-x_i}{x-x_{i+m}}\right)\left(1 - \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{R_{(i+1)\ldots(i+m)} - R_{(i+1)\ldots(i+m-1)}}\right) - 1} \tag{3.2.3}$$

This recurrence generates the rational functions through $m + 1$ points from the ones through m and (the term $R_{(i+1)\ldots(i+m-1)}$ in equation 3.2.3) $m - 1$ points. It is started with

$$R_i = y_i \tag{3.2.4}$$

and with

$$R \equiv [R_{i(i+1)\ldots(i+m)} \quad \text{with} \quad m = -1] = 0 \tag{3.2.5}$$

Now, exactly as in equations (3.1.4) and (3.1.5) above, we can convert the recurrence (3.2.3) to one involving only the small differences

$$\begin{aligned} C_{m,i} &\equiv R_{i\ldots(i+m)} - R_{i\ldots(i+m-1)} \\ D_{m,i} &\equiv R_{i\ldots(i+m)} - R_{(i+1)\ldots(i+m)} \end{aligned} \tag{3.2.6}$$

Note that these satisfy the relation

$$C_{m+1,i} - D_{m+1,i} = C_{m,i+1} - D_{m,i} \tag{3.2.7}$$

which is useful in proving the recurrences

$$\begin{aligned} D_{m+1,i} &= \frac{C_{m,i+1}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}} \\ C_{m+1,i} &= \frac{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i}(C_{m,i+1} - D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}} \end{aligned} \tag{3.2.8}$$

This recurrence is implemented in the following subroutine, whose use is analogous in every way to `polint` in §3.1.

```
SUBROUTINE ratint(xa,ya,n,x,y,dy)
INTEGER n,NMAX
REAL dy,x,y,xa(n),ya(n),TINY
PARAMETER (NMAX=10,TINY=1.e-25)      Largest expected value of n, and a small number.
    Given arrays xa and ya, each of length n, and given a value of x, this routine returns a
    value of y and an accuracy estimate dy. The value returned is that of the diagonal rational
    function, evaluated at x, which passes through the n points (xa_i, ya_i), i = 1...n.
INTEGER i,m,ns
REAL dd,h,hh,t,w,c(NMAX),d(NMAX)
ns=1
hh=abs(x-xa(1))
do 11 i=1,n
    h=abs(x-xa(i))
    if (h.eq.0.)then
        y=ya(i)
        dy=0.0
        return
    else if (h.lt.hh) then
        ns=i
        hh=h
    endif
    c(i)=ya(i)
    d(i)=ya(i)+TINY                  The TINY part is needed to prevent a rare zero-over-
enddo 11                                 zero condition.
y=ya(ns)
ns=ns-1
do 13 m=1,n-1
    do 12 i=1,n-m
```

```
        w=c(i+1)-d(i)
        h=xa(i+m)-x                h will never be zero, since this was tested in the ini-
        t=(xa(i)-x)*d(i)/h             tializing loop.
        dd=t-c(i+1)
        if(dd.eq.0.)pause 'failure in ratint'
          This error condition indicates that the interpolating function has a pole at the re-
          quested value of x.
        dd=w/dd
        d(i)=c(i+1)*dd
        c(i)=t*dd
    enddo 12
    if (2*ns.lt.n-m)then
        dy=c(ns+1)
    else
        dy=d(ns)
        ns=ns-1
    endif
    y=y+dy
enddo 13
return
END
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.2. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 3.

3.3 Cubic Spline Interpolation

Given a tabulated function $y_i = y(x_i),\ i = 1...N$, focus attention on one particular interval, between x_j and x_{j+1}. Linear interpolation in that interval gives the interpolation formula

$$y = Ay_j + By_{j+1} \tag{3.3.1}$$

where

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \qquad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j} \tag{3.3.2}$$

Equations (3.3.1) and (3.3.2) are a special case of the general Lagrange interpolation formula (3.1.1).

Since it is (piecewise) linear, equation (3.3.1) has zero second derivative in the interior of each interval, and an undefined, or infinite, second derivative at the abscissas x_j. The goal of cubic spline interpolation is to get an interpolation formula that is smooth in the first derivative, and continuous in the second derivative, both within an interval and at its boundaries.

Suppose, contrary to fact, that in addition to the tabulated values of y_i, we also have tabulated values for the function's second derivatives, y'', that is, a set

```
        w=c(i+1)-d(i)
        h=xa(i+m)-x                h will never be zero, since this was tested in the ini-
        t=(xa(i)-x)*d(i)/h             tializing loop.
        dd=t-c(i+1)
        if(dd.eq.0.)pause 'failure in ratint'
          This error condition indicates that the interpolating function has a pole at the re-
          quested value of x.
        dd=w/dd
        d(i)=c(i+1)*dd
        c(i)=t*dd
    enddo 12
    if (2*ns.lt.n-m)then
        dy=c(ns+1)
    else
        dy=d(ns)
        ns=ns-1
    endif
    y=y+dy
enddo 13
return
END
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.2. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 3.

3.3 Cubic Spline Interpolation

Given a tabulated function $y_i = y(x_i),\ i = 1...N$, focus attention on one particular interval, between x_j and x_{j+1}. Linear interpolation in that interval gives the interpolation formula

$$y = Ay_j + By_{j+1} \tag{3.3.1}$$

where

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \qquad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j} \tag{3.3.2}$$

Equations (3.3.1) and (3.3.2) are a special case of the general Lagrange interpolation formula (3.1.1).

Since it is (piecewise) linear, equation (3.3.1) has zero second derivative in the interior of each interval, and an undefined, or infinite, second derivative at the abscissas x_j. The goal of cubic spline interpolation is to get an interpolation formula that is smooth in the first derivative, and continuous in the second derivative, both within an interval and at its boundaries.

Suppose, contrary to fact, that in addition to the tabulated values of y_i, we also have tabulated values for the function's second derivatives, y'', that is, a set

of numbers y_i''. Then, within each interval, we can add to the right-hand side of equation (3.3.1) a cubic polynomial whose second derivative varies linearly from a value y_j'' on the left to a value y_{j+1}'' on the right. Doing so, we will have the desired continuous second derivative. If we also construct the cubic polynomial to have zero *values* at x_j and x_{j+1}, then adding it in will not spoil the agreement with the tabulated functional values y_j and y_{j+1} at the endpoints x_j and x_{j+1}.

A little side calculation shows that there is only one way to arrange this construction, namely replacing (3.3.1) by

$$y = Ay_j + By_{j+1} + Cy_j'' + Dy_{j+1}'' \tag{3.3.3}$$

where A and B are defined in (3.3.2) and

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2 \qquad D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2 \tag{3.3.4}$$

Notice that the dependence on the independent variable x in equations (3.3.3) and (3.3.4) is entirely through the linear x-dependence of A and B, and (through A and B) the cubic x-dependence of C and D.

We can readily check that y'' is in fact the second derivative of the new interpolating polynomial. We take derivatives of equation (3.3.3) with respect to x, using the definitions of A, B, C, D to compute $dA/dx, dB/dx, dC/dx$, and dD/dx. The result is

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}'' \tag{3.3.5}$$

for the first derivative, and

$$\frac{d^2y}{dx^2} = Ay_j'' + By_{j+1}'' \tag{3.3.6}$$

for the second derivative. Since $A = 1$ at x_j, $A = 0$ at x_{j+1}, while B is just the other way around, (3.3.6) shows that y'' is just the tabulated second derivative, and also that the second derivative will be continuous across (e.g.) the boundary between the two intervals (x_{j-1}, x_j) and (x_j, x_{j+1}).

The only problem now is that we supposed the y_i'''s to be known, when, actually, they are not. However, we have not yet required that the *first* derivative, computed from equation (3.3.5), be continuous across the boundary between two intervals. The key idea of a cubic spline is to require this continuity and to use it to get equations for the second derivatives y_i''.

The required equations are obtained by setting equation (3.3.5) evaluated for $x = x_j$ in the interval (x_{j-1}, x_j) equal to the same equation evaluated for $x = x_j$ but in the interval (x_j, x_{j+1}). With some rearrangement, this gives (for $j = 2, \ldots, N-1$)

$$\frac{x_j - x_{j-1}}{6}y_{j-1}'' + \frac{x_{j+1} - x_{j-1}}{3}y_j'' + \frac{x_{j+1} - x_j}{6}y_{j+1}'' = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \tag{3.3.7}$$

These are $N - 2$ linear equations in the N unknowns $y_i'', i = 1, \ldots, N$. Therefore there is a two-parameter family of possible solutions.

For a unique solution, we need to specify two further conditions, typically taken as boundary conditions at x_1 and x_N. The most common ways of doing this are either

- set one or both of y_1'' and y_N'' equal to zero, giving the so-called *natural cubic spline*, which has zero second derivative on one or both of its boundaries, or
- set either of y_1'' and y_N'' to values calculated from equation (3.3.5) so as to make the first derivative of the interpolating function have a specified value on either or both boundaries.

One reason that cubic splines are especially practical is that the set of equations (3.3.7), along with the two additional boundary conditions, are not only linear, but also *tridiagonal*. Each y_j'' is coupled only to its nearest neighbors at $j \pm 1$. Therefore, the equations can be solved in $O(N)$ operations by the tridiagonal algorithm (§2.4). That algorithm is concise enough to build right into the spline calculational routine. This makes the routine not completely transparent as an implementation of (3.3.7), so we encourage you to study it carefully, comparing with `tridag` (§2.4).

```
      SUBROUTINE spline(x,y,n,yp1,ypn,y2)
      INTEGER n,NMAX
      REAL yp1,ypn,x(n),y(n),y2(n)
      PARAMETER (NMAX=500)
        Given arrays x(1:n) and y(1:n) containing a tabulated function, i.e., y_i = f(x_i), with
        x_1 < x_2 < ... < x_N, and given values yp1 and ypn for the first derivative of the inter-
        polating function at points 1 and n, respectively, this routine returns an array y2(1:n) of
        length n which contains the second derivatives of the interpolating function at the tabulated
        points x_i. If yp1 and/or ypn are equal to 1 × 10^30 or larger, the routine is signaled to set
        the corresponding boundary condition for a natural spline, with zero second derivative on
        that boundary.
        Parameter: NMAX is the largest anticipated value of n.
      INTEGER i,k
      REAL p,qn,sig,un,u(NMAX)
      if (yp1.gt..99e30) then                  The lower boundary condition is set either to be
        y2(1)=0.                                 "natural"
        u(1)=0.
      else                                     or else to have a specified first derivative.
        y2(1)=-0.5
        u(1)=(3./(x(2)-x(1)))*((y(2)-y(1))/(x(2)-x(1))-yp1)
      endif
      do 11 i=2,n-1                            This is the decomposition loop of the tridiagonal
        sig=(x(i)-x(i-1))/(x(i+1)-x(i-1))        algorithm. y2 and u are used for temporary
        p=sig*y2(i-1)+2.                         storage of the decomposed factors.
        y2(i)=(sig-1.)/p
        u(i)=(6.*((y(i+1)-y(i))/(x(i+1)-x(i))-(y(i)-y(i-1))
     *       /(x(i)-x(i-1)))/(x(i+1)-x(i-1))-sig*u(i-1))/p
      enddo 11
      if (ypn.gt..99e30) then                  The upper boundary condition is set either to be
        qn=0.                                    "natural"
        un=0.
      else                                     or else to have a specified first derivative.
        qn=0.5
        un=(3./(x(n)-x(n-1)))*(ypn-(y(n)-y(n-1))/(x(n)-x(n-1)))
      endif
      y2(n)=(un-qn*u(n-1))/(qn*y2(n-1)+1.)
      do 12 k=n-1,1,-1                         This is the backsubstitution loop of the tridiago-
        y2(k)=y2(k)*y2(k+1)+u(k)                 nal algorithm.
      enddo 12
      return
      END
```

It is important to understand that the program `spline` is called only *once* to process an entire tabulated function in arrays x_i and y_i. Once this has been done,

values of the interpolated function for any value of x are obtained by calls (as many as desired) to a separate routine `splint` (for "*spl*ine *int*erpolation"):

```
      SUBROUTINE splint(xa,ya,y2a,n,x,y)
      INTEGER n
      REAL x,y,xa(n),y2a(n),ya(n)
        Given the arrays xa(1:n) and ya(1:n) of length n, which tabulate a function (with the
        xa_i's in order), and given the array y2a(1:n), which is the output from spline above,
        and given a value of x, this routine returns a cubic-spline interpolated value y.
      INTEGER k,khi,klo
      REAL a,b,h
      klo=1                          We will find the right place in the table by means of bisection.
      khi=n                          This is optimal if sequential calls to this routine are at random
1     if (khi-klo.gt.1) then             values of x. If sequential calls are in order, and closely
        k=(khi+klo)/2                    spaced, one would do better to store previous values of
        if(xa(k).gt.x)then               klo and khi and test if they remain appropriate on the
          khi=k                          next call.
        else
          klo=k
        endif
      goto 1
      endif                          klo and khi now bracket the input value of x.
      h=xa(khi)-xa(klo)
      if (h.eq.0.) pause 'bad xa input in splint'  The xa's must be distinct.
      a=(xa(khi)-x)/h                Cubic spline polynomial is now evaluated.
      b=(x-xa(klo))/h
      y=a*ya(klo)+b*ya(khi)+
     *          ((a**3-a)*y2a(klo)+(b**3-b)*y2a(khi))*(h**2)/6.
      return
      END
```

CITED REFERENCES AND FURTHER READING:

De Boor, C. 1978, *A Practical Guide to Splines* (New York: Springer-Verlag).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §§4.4–4.5.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.4.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §3.8.

3.4 How to Search an Ordered Table

Suppose that you have decided to use some particular interpolation scheme, such as fourth-order polynomial interpolation, to compute a function $f(x)$ from a set of tabulated x_i's and f_i's. Then you will need a fast way of finding your place in the table of x_i's, given some particular value x at which the function evaluation is desired. This problem is not properly one of numerical analysis, but it occurs so often in practice that it would be negligent of us to ignore it.

Formally, the problem is this: Given an array of abscissas `xx(j)`, `j`=1, 2, . . . ,`n`, with the elements either monotonically increasing or monotonically decreasing, and given a number `x`, find an integer `j` such that `x` lies between `xx(j)` and `xx(j+1)`.

values of the interpolated function for any value of x are obtained by calls (as many as desired) to a separate routine `splint` (for "*spl*ine *int*erpolation"):

```
      SUBROUTINE splint(xa,ya,y2a,n,x,y)
      INTEGER n
      REAL x,y,xa(n),y2a(n),ya(n)
        Given the arrays xa(1:n) and ya(1:n) of length n, which tabulate a function (with the
        xa_i's in order), and given the array y2a(1:n), which is the output from spline above,
        and given a value of x, this routine returns a cubic-spline interpolated value y.
      INTEGER k,khi,klo
      REAL a,b,h
      klo=1                          We will find the right place in the table by means of bisection.
      khi=n                          This is optimal if sequential calls to this routine are at random
1     if (khi-klo.gt.1) then             values of x. If sequential calls are in order, and closely
        k=(khi+klo)/2                    spaced, one would do better to store previous values of
        if(xa(k).gt.x)then               klo and khi and test if they remain appropriate on the
          khi=k                          next call.
        else
          klo=k
        endif
      goto 1
      endif                          klo and khi now bracket the input value of x.
      h=xa(khi)-xa(klo)
      if (h.eq.0.) pause 'bad xa input in splint'  The xa's must be distinct.
      a=(xa(khi)-x)/h                Cubic spline polynomial is now evaluated.
      b=(x-xa(klo))/h
      y=a*ya(klo)+b*ya(khi)+
     *          ((a**3-a)*y2a(klo)+(b**3-b)*y2a(khi))*(h**2)/6.
      return
      END
```

CITED REFERENCES AND FURTHER READING:

De Boor, C. 1978, *A Practical Guide to Splines* (New York: Springer-Verlag).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §§4.4–4.5.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.4.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §3.8.

3.4 How to Search an Ordered Table

Suppose that you have decided to use some particular interpolation scheme, such as fourth-order polynomial interpolation, to compute a function $f(x)$ from a set of tabulated x_i's and f_i's. Then you will need a fast way of finding your place in the table of x_i's, given some particular value x at which the function evaluation is desired. This problem is not properly one of numerical analysis, but it occurs so often in practice that it would be negligent of us to ignore it.

Formally, the problem is this: Given an array of abscissas `xx(j)`, `j`=1, 2, . . . ,`n`, with the elements either monotonically increasing or monotonically decreasing, and given a number `x`, find an integer `j` such that `x` lies between `xx(j)` and `xx(j+1)`.

For this task, let us define fictitious array elements xx(0) and xx(n+1) equal to plus or minus infinity (in whichever order is consistent with the monotonicity of the table). Then j will always be between 0 and n, inclusive; a returned value of 0 indicates "off-scale" at one end of the table, n indicates off-scale at the other end.

In most cases, when all is said and done, it is hard to do better than *bisection*, which will find the right place in the table in about $\log_2$n tries. We already did use bisection in the spline evaluation routine splint of the preceding section, so you might glance back at that. Standing by itself, a bisection routine looks like this:

```
      SUBROUTINE locate(xx,n,x,j)
      INTEGER j,n
      REAL x,xx(n)
         Given an array xx(1:n), and given a value x, returns a value j such that x is between
         xx(j) and xx(j+1). xx(1:n) must be monotonic, either increasing or decreasing. j=0
         or j=n is returned to indicate that x is out of range.
      INTEGER jl,jm,ju
      jl=0                            Initialize lower
      ju=n+1                          and upper limits.
10    if(ju-jl.gt.1)then              If we are not yet done,
          jm=(ju+jl)/2                compute a midpoint,
          if((xx(n).ge.xx(1)).eqv.(x.ge.xx(jm)))then
              jl=jm                   and replace either the lower limit
          else
              ju=jm                   or the upper limit, as appropriate.
          endif
      goto 10                         Repeat until
      endif                           the test condition 10 is satisfied.
      if(x.eq.xx(1))then              Then set the output
          j=1
      else if(x.eq.xx(n))then
          j=n-1
      else
          j=jl
      endif
      return                          and return.
      END
```

Note the use of the logical equality relation .eqv., which is true when its two logical operands are either both true or both false. This relation allows the routine to work for both monotonically increasing and monotonically decreasing orders of xx(1:n).

Search with Correlated Values

Sometimes you will be in the situation of searching a large table many times, and with nearly identical abscissas on consecutive searches. For example, you may be generating a function that is used on the right-hand side of a differential equation: Most differential-equation integrators, as we shall see in Chapter 16, call for right-hand side evaluations at points that hop back and forth a bit, but whose trend moves slowly in the direction of the integration.

In such cases it is wasteful to do a full bisection, *ab initio*, on each call. The following routine instead starts with a guessed position in the table. It first "hunts," either up or down, in increments of 1, then 2, then 4, etc., until the desired value is bracketed. Second, it then bisects in the bracketed interval. At worst, this routine is about a factor of 2 slower than locate above (if the hunt phase expands to include the whole table). At best, it can be a factor of $\log_2$n faster than locate, if the desired point is usually quite close to the input guess. Figure 3.4.1 compares the two routines.

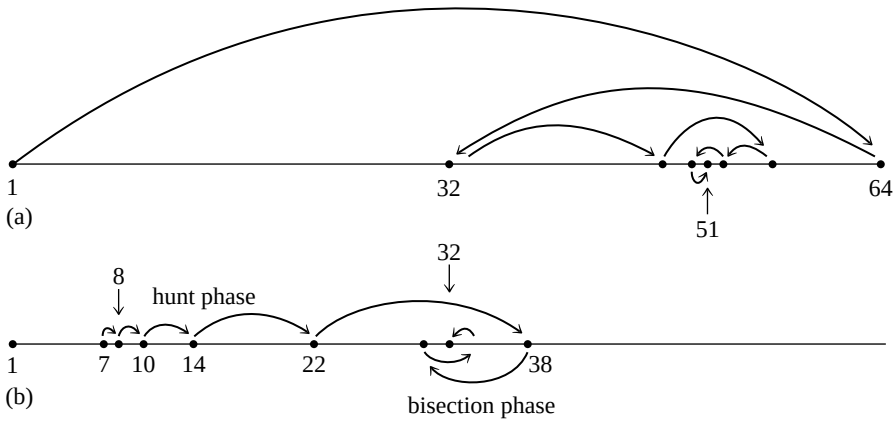


Figure 3.4.1. (a) The routine `locate` finds a table entry by bisection. Shown here is the sequence of steps that converge to element 51 in a table of length 64. (b) The routine `hunt` searches from a previous known position in the table by increasing steps, then converges by bisection. Shown here is a particularly unfavorable example, converging to element 32 from element 7. A favorable example would be convergence to an element near 7, such as 9, which would require just three "hops."

```
      SUBROUTINE hunt(xx,n,x,jlo)
      INTEGER jlo,n
      REAL x,xx(n)
        Given an array xx(1:n), and given a value x, returns a value jlo such that x is between
        xx(jlo) and xx(jlo+1). xx(1:n) must be monotonic, either increasing or decreasing.
        jlo=0 or jlo=n is returned to indicate that x is out of range. jlo on input is taken as
        the initial guess for jlo on output.
      INTEGER inc,jhi,jm
      LOGICAL ascnd
      ascnd=xx(n).ge.xx(1)                    True if ascending order of table, false otherwise.
      if(jlo.le.0.or.jlo.gt.n)then            Input guess not useful. Go immediately to bisection.
        jlo=0
        jhi=n+1
        goto 3
      endif
      inc=1                                   Set the hunting increment.
      if(x.ge.xx(jlo).eqv.ascnd)then          Hunt up:
1       jhi=jlo+inc
        if(jhi.gt.n)then                      Done hunting, since off end of table.
          jhi=n+1
        else if(x.ge.xx(jhi).eqv.ascnd)then       Not done hunting,
          jlo=jhi
          inc=inc+inc                         so double the increment
          goto 1                              and try again.
        endif                                 Done hunting, value bracketed.
      else                                    Hunt down:
        jhi=jlo
2       jlo=jhi-inc
        if(jlo.lt.1)then                      Done hunting, since off end of table.
          jlo=0
        else if(x.lt.xx(jlo).eqv.ascnd)then       Not done hunting,
          jhi=jlo
          inc=inc+inc                         so double the increment
          goto 2                              and try again.
        endif                                 Done hunting, value bracketed.
      endif                                   Hunt is done, so begin the final bisection phase:
3     if(jhi-jlo.eq.1)then
        if(x.eq.xx(n))jlo=n-1
        if(x.eq.xx(1))jlo=1
```

```
    return
endif
jm=(jhi+jlo)/2
if(x.ge.xx(jm).eqv.ascnd)then
    jlo=jm
else
    jhi=jm
endif
goto 3
END
```

After the Hunt

The problem: Routines `locate` and `hunt` return an index `j` such that your desired value lies between table entries `xx(j)` and `xx(j+1)`, where `xx(1:n)` is the full length of the table. But, to obtain an `m`-point interpolated value using a routine like `polint` (§3.1) or `ratint` (§3.2), you need to supply much shorter `xx` and `yy` arrays, of length `m`. How do you make the connection?

The solution: Calculate

$$\texttt{k} = \texttt{min(max(j-(m-1)/2,1),n+1-m)}$$

This expression produces the index of the leftmost member of an `m`-point set of points centered (insofar as possible) between `j` and `j+1`, but bounded by 1 at the left and `n` at the right. `FORTRAN` then lets you call the interpolation routine with array addresses offset by `k`, e.g.,

```
call polint(xx(k),yy(k),m,...)
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.2.1.

3.5 Coefficients of the Interpolating Polynomial

Occasionally you may wish to know not the value of the interpolating polynomial that passes through a (small!) number of points, but the coefficients of that polynomial. A valid use of the coefficients might be, for example, to compute simultaneous interpolated values of the function and of several of its derivatives (see §5.3), or to convolve a segment of the tabulated function with some other function, where the moments of that other function (i.e., its convolution with powers of x) are known analytically.

However, please be certain that the coefficients are what you need. Generally the coefficients of the interpolating polynomial can be determined much less accurately than its value at a desired abscissa. Therefore it is not a good idea to determine the coefficients only for use in calculating interpolating values. Values thus calculated will not pass exactly through the tabulated points, for example, while values computed by the routines in §3.1–§3.3 will pass exactly through such points.

```
    return
endif
jm=(jhi+jlo)/2
if(x.ge.xx(jm).eqv.ascnd)then
    jlo=jm
else
    jhi=jm
endif
goto 3
END
```

After the Hunt

The problem: Routines `locate` and `hunt` return an index `j` such that your desired value lies between table entries `xx(j)` and `xx(j+1)`, where `xx(1:n)` is the full length of the table. But, to obtain an `m`-point interpolated value using a routine like `polint` (§3.1) or `ratint` (§3.2), you need to supply much shorter `xx` and `yy` arrays, of length `m`. How do you make the connection?

The solution: Calculate

$$\texttt{k} = \texttt{min(max(j-(m-1)/2,1),n+1-m)}$$

This expression produces the index of the leftmost member of an `m`-point set of points centered (insofar as possible) between `j` and `j+1`, but bounded by 1 at the left and `n` at the right. `FORTRAN` then lets you call the interpolation routine with array addresses offset by `k`, e.g.,

```
call polint(xx(k),yy(k),m,...)
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.2.1.

3.5 Coefficients of the Interpolating Polynomial

Occasionally you may wish to know not the value of the interpolating polynomial that passes through a (small!) number of points, but the coefficients of that polynomial. A valid use of the coefficients might be, for example, to compute simultaneous interpolated values of the function and of several of its derivatives (see §5.3), or to convolve a segment of the tabulated function with some other function, where the moments of that other function (i.e., its convolution with powers of x) are known analytically.

However, please be certain that the coefficients are what you need. Generally the coefficients of the interpolating polynomial can be determined much less accurately than its value at a desired abscissa. Therefore it is not a good idea to determine the coefficients only for use in calculating interpolating values. Values thus calculated will not pass exactly through the tabulated points, for example, while values computed by the routines in §3.1–§3.3 will pass exactly through such points.

Also, you should not mistake the interpolating polynomial (and its coefficients) for its cousin, the *best fit* polynomial through a data set. Fitting is a *smoothing* process, since the number of fitted coefficients is typically much less than the number of data points. Therefore, fitted coefficients can be accurately and stably determined even in the presence of statistical errors in the tabulated values. (See §14.8.) Interpolation, where the number of coefficients and number of tabulated points are equal, takes the tabulated values as perfect. If they in fact contain statistical errors, these can be magnified into oscillations of the interpolating polynomial in between the tabulated points.

As before, we take the tabulated points to be $y_i \equiv y(x_i)$. If the interpolating polynomial is written as

$$y = c_1 + c_2 x + c_3 x^2 + \cdots + c_N x^{N-1} \tag{3.5.1}$$

then the c_i's are required to satisfy the linear equation

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{3.5.2}$$

This is a *Vandermonde matrix*, as described in §2.8. One could in principle solve equation (3.5.2) by standard techniques for linear equations generally (§2.3); however the special method that was derived in §2.8 is more efficient by a large factor, of order N, so it is much better.

Remember that Vandermonde systems can be quite ill-conditioned. In such a case, *no* numerical method is going to give a very accurate answer. Such cases do not, please note, imply any difficulty in finding interpolated *values* by the methods of §3.1, but only difficulty in finding *coefficients*.

Like the routine in §2.8, the following is due to G.B. Rybicki.

```
SUBROUTINE polcoe(x,y,n,cof)
INTEGER n,NMAX
REAL cof(n),x(n),y(n)
PARAMETER (NMAX=15)                    Largest anticipated value of n.
    Given arrays x(1:n) and y(1:n) containing a tabulated function y_i = f(x_i), this routine
    returns an array of coefficients cof(1:n), such that y_i = sum_j cof_j x_i^{j-1}.
INTEGER i,j,k
REAL b,ff,phi,s(NMAX)
do 11 i=1,n
    s(i)=0.
    cof(i)=0.
enddo 11
s(n)=-x(1)
do 13 i=2,n                            Coefficients s_i of the master polynomial P(x) are found
    do 12 j=n+1-i,n-1                      by recurrence.
        s(j)=s(j)-x(i)*s(j+1)
    enddo 12
    s(n)=s(n)-x(i)
enddo 13
do 16 j=1,n
    phi=n
    do 14 k=n-1,1,-1                   The quantity phi = prod_{j!=k}(x_j - x_k) is found as a deriva-
                                           tive of P(x_j).
```

```
        phi=k*s(k+1)+x(j)*phi
    enddo 14
    ff=y(j)/phi
    b=1.                          Coefficients of polynomials in each term of the Lagrange
    do 15 k=n,1,-1                    formula are found by synthetic division of P(x) by
        cof(k)=cof(k)+b*ff            (x − x_j). The solution c_k is accumulated.
        b=s(k)+x(j)*b
    enddo 15
enddo 16
return
END
```

Another Method

Another technique is to make use of the function value interpolation routine already given (`polint` §3.1). If we interpolate (or extrapolate) to find the value of the interpolating polynomial at $x = 0$, then this value will evidently be c_1. Now we can subtract c_1 from the y_i's and divide each by its corresponding x_i. Throwing out one point (the one with smallest x_i is a good candidate), we can repeat the procedure to find c_2, and so on.

It is not instantly obvious that this procedure is stable, but we have generally found it to be somewhat *more* stable than the routine immediately preceding. This method is of order N^3, while the preceding one was of order N^2. You will find, however, that neither works very well for large N, because of the intrinsic ill-condition of the Vandermonde problem. In single precision, N up to 8 or 10 is satisfactory; about double this in double precision.

```
SUBROUTINE polcof(xa,ya,n,cof)
INTEGER n,NMAX
REAL cof(n),xa(n),ya(n)
PARAMETER (NMAX=15)                          Largest anticipated value of n.
C  USES polint
    Given arrays xa(1:n) and ya(1:n) of length n containing a tabulated function ya_i =
    f(xa_i), this routine returns an array of coefficients cof(1:n), also of length n, such that
    ya_i = Σ_j cof_j xa_i^(j−1).
INTEGER i,j,k
REAL dy,xmin,x(NMAX),y(NMAX)
do 11 j=1,n
    x(j)=xa(j)
    y(j)=ya(j)
enddo 11
do 14 j=1,n
    call polint(x,y,n+1-j,0.,cof(j),dy)      This is the polynomial interpolation rou-
    xmin=1.e38                                   tine of §3.1. We extrapolate to x =
    k=0                                          0.
    do 12 i=1,n+1-j                          Find the remaining x_i of smallest abso-
        if (abs(x(i)).lt.xmin)then               lute value,
            xmin=abs(x(i))
            k=i
        endif
        if(x(i).ne.0.)y(i)=(y(i)-cof(j))/x(i)    (meanwhile reducing all the terms)
    enddo 12
    do 13 i=k+1,n+1-j                            and eliminate it.
        y(i-1)=y(i)
        x(i-1)=x(i)
    enddo 13
enddo 14
return
END
```

If the point $x = 0$ is not in (or at least close to) the range of the tabulated x_i's, then the coefficients of the interpolating polynomial will in general become very large. However, the real "information content" of the coefficients is in small differences from the "translation-induced" large values. This is one cause of ill-conditioning, resulting in loss of significance and poorly determined coefficients. You should consider redefining the origin of the problem, to put $x = 0$ in a sensible place.

Another pathology is that, if too high a degree of interpolation is attempted on a smooth function, the interpolating polynomial will attempt to use its high-degree coefficients, in combinations with large and almost precisely canceling combinations, to match the tabulated values down to the last possible epsilon of accuracy. This effect is the same as the intrinsic tendency of the interpolating polynomial values to oscillate (wildly) between its constrained points, and would be present even if the machine's floating precision were infinitely good. The above routines `polcoe` and `polcof` have slightly different sensitivities to the pathologies that can occur.

Are you still quite certain that using the *coefficients* is a good idea?

CITED REFERENCES AND FURTHER READING:
Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §5.2.

3.6 Interpolation in Two or More Dimensions

In multidimensional interpolation, we seek an estimate of $y(x_1, x_2, \ldots, x_n)$ from an n-dimensional grid of tabulated values y and n one-dimensional vectors giving the tabulated values of each of the independent variables $x_1, x_2, \ldots, x_n$. We will not here consider the problem of interpolating on a mesh that is not Cartesian, i.e., has tabulated function values at "random" points in n-dimensional space rather than at the vertices of a rectangular array. For clarity, we will consider explicitly only the case of two dimensions, the cases of three or more dimensions being analogous in every way.

In two dimensions, we imagine that we are given a matrix of functional values `ya(j,k)`, where `j` varies from 1 to `m`, and `k` varies from 1 to `n`. We are also given an array `x1a` of length `m`, and an array `x2a` of length `n`. The relation of these input quantities to an underlying function $y(x_1, x_2)$ is

$$\texttt{ya(j,k)} = y(\texttt{x1a(j)}, \texttt{x2a(k)}) \tag{3.6.1}$$

We want to estimate, by interpolation, the function y at some untabulated point (x_1, x_2).

An important concept is that of the *grid square* in which the point (x_1, x_2) falls, that is, the four tabulated points that surround the desired interior point. For convenience, we will number these points from 1 to 4, counterclockwise starting from the lower left (see Figure 3.6.1). More precisely, if

$$\begin{aligned} \texttt{x1a(j)} \le x_1 \le \texttt{x1a(j+1)} \\ \texttt{x2a(k)} \le x_2 \le \texttt{x2a(k+1)} \end{aligned} \tag{3.6.2}$$

If the point $x = 0$ is not in (or at least close to) the range of the tabulated x_i's, then the coefficients of the interpolating polynomial will in general become very large. However, the real "information content" of the coefficients is in small differences from the "translation-induced" large values. This is one cause of ill-conditioning, resulting in loss of significance and poorly determined coefficients. You should consider redefining the origin of the problem, to put $x = 0$ in a sensible place.

Another pathology is that, if too high a degree of interpolation is attempted on a smooth function, the interpolating polynomial will attempt to use its high-degree coefficients, in combinations with large and almost precisely canceling combinations, to match the tabulated values down to the last possible epsilon of accuracy. This effect is the same as the intrinsic tendency of the interpolating polynomial values to oscillate (wildly) between its constrained points, and would be present even if the machine's floating precision were infinitely good. The above routines `polcoe` and `polcof` have slightly different sensitivities to the pathologies that can occur.

Are you still quite certain that using the *coefficients* is a good idea?

CITED REFERENCES AND FURTHER READING:
Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §5.2.

3.6 Interpolation in Two or More Dimensions

In multidimensional interpolation, we seek an estimate of $y(x_1, x_2, \ldots, x_n)$ from an n-dimensional grid of tabulated values y and n one-dimensional vectors giving the tabulated values of each of the independent variables $x_1, x_2, \ldots, x_n$. We will not here consider the problem of interpolating on a mesh that is not Cartesian, i.e., has tabulated function values at "random" points in n-dimensional space rather than at the vertices of a rectangular array. For clarity, we will consider explicitly only the case of two dimensions, the cases of three or more dimensions being analogous in every way.

In two dimensions, we imagine that we are given a matrix of functional values `ya(j,k)`, where `j` varies from 1 to `m`, and `k` varies from 1 to `n`. We are also given an array `x1a` of length `m`, and an array `x2a` of length `n`. The relation of these input quantities to an underlying function $y(x_1, x_2)$ is

$$\texttt{ya(j,k)} = y(\texttt{x1a(j)}, \texttt{x2a(k)}) \tag{3.6.1}$$

We want to estimate, by interpolation, the function y at some untabulated point (x_1, x_2).

An important concept is that of the *grid square* in which the point (x_1, x_2) falls, that is, the four tabulated points that surround the desired interior point. For convenience, we will number these points from 1 to 4, counterclockwise starting from the lower left (see Figure 3.6.1). More precisely, if

$$\begin{aligned} \texttt{x1a(j)} \le x_1 \le \texttt{x1a(j+1)} \\ \texttt{x2a(k)} \le x_2 \le \texttt{x2a(k+1)} \end{aligned} \tag{3.6.2}$$

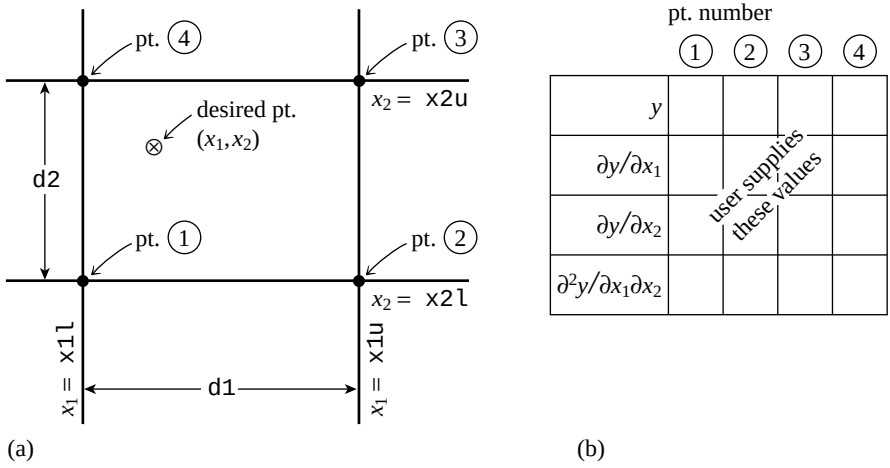


Figure 3.6.1. (a) Labeling of points used in the two-dimensional interpolation routines `bcuint` and `bcucof`. (b) For each of the four points in (a), the user supplies one function value, two first derivatives, and one cross-derivative, a total of 16 numbers.

defines `j` and `k`, then

$$\begin{aligned} y_1 &\equiv \texttt{ya(j,k)} \\ y_2 &\equiv \texttt{ya(j+1,k)} \\ y_3 &\equiv \texttt{ya(j+1,k+1)} \\ y_4 &\equiv \texttt{ya(j,k+1)} \end{aligned} \tag{3.6.3}$$

The simplest interpolation in two dimensions is *bilinear interpolation* on the grid square. Its formulas are:

$$\begin{aligned} t &\equiv (x_1 - \texttt{x1a(j)})/(\texttt{x1a(j+1)} - \texttt{x1a(j)}) \\ u &\equiv (x_2 - \texttt{x2a(k)})/(\texttt{x2a(k+1)} - \texttt{x2a(k)}) \end{aligned} \tag{3.6.4}$$

(so that t and u each lie between 0 and 1), and

$$y(x_1, x_2) = (1-t)(1-u)y_1 + t(1-u)y_2 + tuy_3 + (1-t)uy_4 \tag{3.6.5}$$

Bilinear interpolation is frequently "close enough for government work." As the interpolating point wanders from grid square to grid square, the interpolated function value changes continuously. However, the gradient of the interpolated function changes discontinuously at the boundaries of each grid square.

There are two distinctly different directions that one can take in going beyond bilinear interpolation to higher-order methods: One can use higher order to obtain increased accuracy for the interpolated function (for sufficiently smooth functions!), without necessarily trying to fix up the continuity of the gradient and higher derivatives. Or, one can make use of higher order to enforce smoothness of some of these derivatives as the interpolating point crosses grid-square boundaries. We will now consider each of these two directions in turn.

Higher Order for Accuracy

The basic idea is to break up the problem into a succession of one-dimensional interpolations. If we want to do `m-1` order interpolation in the x_1 direction, and `n-1` order in the x_2 direction, we first locate an `m` × `n` sub-block of the tabulated function matrix that contains our desired point (x_1, x_2). We then do `m` one-dimensional interpolations in the x_2 direction, i.e., on the rows of the sub-block, to get function values at the points $(\texttt{x1a(j)}, x_2)$, $\texttt{j} = 1, \ldots, \texttt{m}$. Finally, we do a last interpolation in the x_1 direction to get the answer. If we use the polynomial interpolation routine `polint` of §3.1, and a sub-block which is presumed to be already located (and copied into an `m` by `n` array `ya`), the procedure looks like this:

```
      SUBROUTINE polin2(x1a,x2a,ya,m,n,x1,x2,y,dy)
      INTEGER m,n,NMAX,MMAX
      REAL dy,x1,x2,y,x1a(m),x2a(n),ya(m,n)
      PARAMETER (NMAX=20,MMAX=20)                    Maximum expected values of n and m.
C     USES polint
        Given arrays x1a(1:m) and x2a(1:n) of independent variables, and an m by n array of
        function values ya(1:m,1:n), tabulated at the grid points defined by x1a and x2a; and
        given values x1 and x2 of the independent variables; this routine returns an interpolated
        function value y, and an accuracy indication dy (based only on the interpolation in the x1
        direction, however).
      INTEGER j,k
      REAL ymtmp(MMAX),yntmp(NMAX)
      do 12 j=1,m                                    Loop over rows.
        do 11 k=1,n                                  Copy the row into temporary storage.
          yntmp(k)=ya(j,k)
        enddo 11
        call polint(x2a,yntmp,n,x2,ymtmp(j),dy)      Interpolate answer into temporary stor-
      enddo 12                                           age.
      call polint(x1a,ymtmp,m,x1,y,dy)               Do the final interpolation.
      return
      END
```

Higher Order for Smoothness: Bicubic Interpolation

We will give two methods that are in common use, and which are themselves not unrelated. The first is usually called *bicubic interpolation.*

Bicubic interpolation requires the user to specify at each grid point not just the function $y(x_1, x_2)$, but also the gradients $\partial y/\partial x_1 \equiv y_{,1}$, $\partial y/\partial x_2 \equiv y_{,2}$ and the cross derivative $\partial^2 y/\partial x_1 \partial x_2 \equiv y_{,12}$. Then an interpolating function that is *cubic* in the scaled coordinates t and u (equation 3.6.4) can be found, with the following properties: (i) The values of the function and the specified derivatives are reproduced exactly on the grid points, and (ii) the values of the function and the specified derivatives change continuously as the interpolating point crosses from one grid square to another.

It is important to understand that nothing in the equations of bicubic interpolation requires you to specify the extra derivatives *correctly*! The smoothness properties are tautologically "forced," and have nothing to do with the "accuracy" of the specified derivatives. It is a separate problem for you to decide how to obtain the values that are specified. The better you do, the more *accurate* the interpolation will be. But it will be *smooth* no matter what you do.

Best of all is to know the derivatives analytically, or to be able to compute them accurately by numerical means, at the grid points. Next best is to determine them by numerical differencing from the functional values already tabulated on the grid. The relevant code would be something like this (using centered differencing):

```
y1a(j,k)=(ya(j+1,k)-ya(j-1,k))/(x1a(j+1)-x1a(j-1))
y2a(j,k)=(ya(j,k+1)-ya(j,k-1))/(x2a(k+1)-x2a(k-1))
y12a(j,k)=(ya(j+1,k+1)-ya(j+1,k-1)-ya(j-1,k+1)+ya(j-1,k-1))
          /((x1a(j+1)-x1a(j-1))*(x2a(k+1)-x2a(k-1)))
```

To do a bicubic interpolation within a grid square, given the function y and the derivatives y1, y2, y12 at each of the four corners of the square, there are two steps: First obtain the sixteen quantities $c_{ij},\ i,j = 1,\ldots,4$ using the routine bcucof below. (The formulas that obtain the c's from the function and derivative values are just a complicated linear transformation, with coefficients which, having been determined once in the mists of numerical history, can be tabulated and forgotten.) Next, substitute the c's into any or all of the following bicubic formulas for function and derivatives, as desired:

$$\begin{aligned} y(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} c_{ij}t^{i-1}u^{j-1} \\ y_{,1}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4}(i-1)c_{ij}t^{i-2}u^{j-1}(dt/dx_1) \\ y_{,2}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4}(j-1)c_{ij}t^{i-1}u^{j-2}(du/dx_2) \\ y_{,12}(x_1,x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4}(i-1)(j-1)c_{ij}t^{i-2}u^{j-2}(dt/dx_1)(du/dx_2) \end{aligned} \tag{3.6.6}$$

where t and u are again given by equation (3.6.4).

```
      SUBROUTINE bcucof(y,y1,y2,y12,d1,d2,c)
      REAL d1,d2,c(4,4),y(4),y1(4),y12(4),y2(4)
          Given arrays y,y1,y2, and y12, each of length 4, containing the function, gradients, and
          cross derivative at the four grid points of a rectangular grid cell (numbered counterclockwise
          from the lower left), and given d1 and d2, the length of the grid cell in the 1- and 2-
          directions, this routine returns the table c(1:4,1:4) that is used by routine bcuint for
          bicubic interpolation.
      INTEGER i,j,k,l
      REAL d1d2,xx,cl(16),wt(16,16),x(16)
      SAVE wt
      DATA wt/1,0,-3,2,4*0,-3,0,9,-6,2,0,-6,4,8*0,3,0,-9,6,-2,0,6,-4
     *  ,10*0,9,-6,2*0,-6,4,2*0,3,-2,6*0,-9,6,2*0,6,-4
     *  ,4*0,1,0,-3,2,-2,0,6,-4,1,0,-3,2,8*0,-1,0,3,-2,1,0,-3,2
     *  ,10*0,-3,2,2*0,3,-2,6*0,3,-2,2*0,-6,4,2*0,3,-2
     *  ,0,1,-2,1,5*0,-3,6,-3,0,2,-4,2,9*0,3,-6,3,0,-2,4,-2
     *  ,10*0,-3,3,2*0,2,-2,2*0,-1,1,6*0,3,-3,2*0,-2,2
     *  ,5*0,1,-2,1,0,-2,4,-2,0,1,-2,1,9*0,-1,2,-1,0,1,-2,1
     *  ,10*0,1,-1,2*0,-1,1,6*0,-1,1,2*0,2,-2,2*0,-1,1/
      d1d2=d1*d2
      do 11 i=1,4                    Pack a temporary vector x.
          x(i)=y(i)
```

```
        x(i+4)=y1(i)*d1
        x(i+8)=y2(i)*d2
        x(i+12)=y12(i)*d1d2
    enddo 11
    do 13 i=1,16                    Matrix multiply by the stored table.
        xx=0.
        do 12 k=1,16
            xx=xx+wt(i,k)*x(k)
        enddo 12
        cl(i)=xx
    enddo 13
    l=0
    do 15 i=1,4                     Unpack the result into the output table.
        do 14 j=1,4
            l=l+1
            c(i,j)=cl(l)
        enddo 14
    enddo 15
    return
    END
```

The implementation of equation (3.6.6), which performs a bicubic interpolation, returns the interpolated function value and the two gradient values, and uses the above routine `bcucof`, is simply:

```
    SUBROUTINE bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,
*       ansy1,ansy2)
    REAL ansy,ansy1,ansy2,x1,x1l,x1u,x2,x2l,x2u,y(4),y1(4),
*       y12(4),y2(4)
C   USES bcucof
```

Bicubic interpolation within a grid square. Input quantities are `y,y1,y2,y12` (as described in `bcucof`); `x1l` and `x1u`, the lower and upper coordinates of the grid square in the 1-direction; `x2l` and `x2u` likewise for the 2-direction; and `x1,x2`, the coordinates of the desired point for the interpolation. The interpolated function value is returned as `ansy`, and the interpolated gradient values as `ansy1` and `ansy2`. This routine calls `bcucof`.

```
    INTEGER i
    REAL t,u,c(4,4)
    call bcucof(y,y1,y2,y12,x1u-x1l,x2u-x2l,c)      Get the c's.
    if(x1u.eq.x1l.or.x2u.eq.x2l)pause 'bad input in bcuint'
    t=(x1-x1l)/(x1u-x1l)                            Equation (3.6.4).
    u=(x2-x2l)/(x2u-x2l)
    ansy=0.
    ansy2=0.
    ansy1=0.
    do 11 i=4,1,-1                                  Equation (3.6.6).
        ansy=t*ansy+((c(i,4)*u+c(i,3))*u+c(i,2))*u+c(i,1)
        ansy2=t*ansy2+(3.*c(i,4)*u+2.*c(i,3))*u+c(i,2)
        ansy1=u*ansy1+(3.*c(4,i)*t+2.*c(3,i))*t+c(2,i)
    enddo 11
    ansy1=ansy1/(x1u-x1l)
    ansy2=ansy2/(x2u-x2l)
    return
    END
```

Higher Order for Smoothness: Bicubic Spline

The other common technique for obtaining smoothness in two-dimensional interpolation is the *bicubic spline*. Actually, this is equivalent to a special case

of bicubic interpolation: The interpolating function is of the same functional form as equation (3.6.6); the values of the derivatives at the grid points are, however, determined "globally" by one-dimensional splines. However, bicubic splines are usually implemented in a form that looks rather different from the above bicubic interpolation routines, instead looking much closer in form to the routine polin2 above: To interpolate one functional value, one performs m one-dimensional splines across the rows of the table, followed by one additional one-dimensional spline down the newly created column. It is a matter of taste (and trade-off between time and memory) as to how much of this process one wants to precompute and store. Instead of precomputing and storing all the derivative information (as in bicubic interpolation), spline users typically precompute and store only one auxiliary table, of second derivatives in one direction only. Then one need only do spline *evaluations* (not constructions) for the m row splines; one must still do a construction *and* an evaluation for the final column spline. (Recall that a spline construction is a process of order N, while a spline evaluation is only of order $\log N$ — and that is just to find the place in the table!)

Here is a routine to precompute the auxiliary second-derivative table:

```
      SUBROUTINE splie2(x1a,x2a,ya,m,n,y2a)
      INTEGER m,n,NN
      REAL x1a(m),x2a(n),y2a(m,n),ya(m,n)
      PARAMETER (NN=100)                                  Maximum expected value of n and m.
C     USES spline
        Given an m by n tabulated function ya(1:m,1:n), and tabulated independent variables
        x2a(1:n), this routine constructs one-dimensional natural cubic splines of the rows of ya
        and returns the second-derivatives in the array y2a(1:m,1:n). (The array x1a is included
        in the argument list merely for consistency with routine splin2.)
      INTEGER j,k
      REAL y2tmp(NN),ytmp(NN)
      do 13 j=1,m
        do 11 k=1,n
          ytmp(k)=ya(j,k)
        enddo 11
        call spline(x2a,ytmp,n,1.e30,1.e30,y2tmp)     Values 1×10^30 signal a natural spline.
        do 12 k=1,n
          y2a(j,k)=y2tmp(k)
        enddo 12
      enddo 13
      return
      END
```

After the above routine has been executed once, any number of bicubic spline interpolations can be performed by successive calls of the following routine:

```
      SUBROUTINE splin2(x1a,x2a,ya,y2a,m,n,x1,x2,y)
      INTEGER m,n,NN
      REAL x1,x2,y,x1a(m),x2a(n),y2a(m,n),ya(m,n)
      PARAMETER (NN=100)              Maximum expected value of n and m.
C     USES spline,splint
        Given x1a, x2a, ya, m, n as described in splie2 and y2a as produced by that routine;
        and given a desired interpolating point x1,x2; this routine returns an interpolated function
        value y by bicubic spline interpolation.
      INTEGER j,k
      REAL y2tmp(NN),ytmp(NN),yytmp(NN)
```

```
do 12 j=1,m                                          Perform m evaluations of the row splines
    do 11 k=1,n                                        constructed by splie2, using the one-
        ytmp(k)=ya(j,k)                                dimensional spline evaluator splint.
        y2tmp(k)=y2a(j,k)
    enddo 11
    call splint(x2a,ytmp,y2tmp,n,x2,yytmp(j))
enddo 12
call spline(x1a,yytmp,m,1.e30,1.e30,y2tmp)           Construct the one-dimensional column spline
call splint(x1a,yytmp,y2tmp,m,x1,y)                    and evaluate it.
return
END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Kinahan, B.F., and Harm, R. 1975, *Astrophysical Journal*, vol. 200, pp. 330–335.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §5.2.7.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.7.

Chapter 4. Integration of Functions

4.0 Introduction

Numerical integration, which is also called *quadrature*, has a history extending back to the invention of calculus and before. The fact that integrals of elementary functions could not, in general, be computed analytically, while derivatives *could* be, served to give the field a certain panache, and to set it a cut above the arithmetic drudgery of numerical analysis during the whole of the 18th and 19th centuries.

With the invention of automatic computing, quadrature became just one numerical task among many, and not a very interesting one at that. Automatic computing, even the most primitive sort involving desk calculators and rooms full of "computers" (that were, until the 1950s, people rather than machines), opened to feasibility the much richer field of numerical integration of differential equations. Quadrature is merely the simplest special case: The evaluation of the integral

$$I = \int_a^b f(x)dx \tag{4.0.1}$$

is precisely equivalent to solving for the value $I \equiv y(b)$ the differential equation

$$\frac{dy}{dx} = f(x) \tag{4.0.2}$$

with the boundary condition

$$y(a) = 0 \tag{4.0.3}$$

Chapter 16 of this book deals with the numerical integration of differential equations. In that chapter, much emphasis is given to the concept of "variable" or "adaptive" choices of stepsize. We will not, therefore, develop that material here. If the function that you propose to integrate is sharply concentrated in one or more peaks, or if its shape is not readily characterized by a single length-scale, then it is likely that you should cast the problem in the form of (4.0.2)–(4.0.3) and use the methods of Chapter 16.

The quadrature methods in this chapter are based, in one way or another, on the obvious device of adding up the value of the integrand at a sequence of abscissas within the range of integration. The game is to obtain the integral as accurately as possible with the smallest number of function evaluations of the integrand. Just as in the case of interpolation (Chapter 3), one has the freedom to choose methods

of various *orders*, with higher order sometimes, but not always, giving higher accuracy. "Romberg integration," which is discussed in §4.3, is a general formalism for making use of integration methods of a variety of different orders, and we recommend it highly.

Apart from the methods of this chapter and of Chapter 16, there are yet other methods for obtaining integrals. One important class is based on function approximation. We discuss explicitly the integration of functions by Chebyshev approximation ("Clenshaw-Curtis" quadrature) in §5.9. Although not explicitly discussed here, you ought to be able to figure out how to do *cubic spline quadrature* using the output of the routine `spline` in §3.3. (Hint: Integrate equation 3.3.3 over x analytically. See [1].)

Some integrals related to Fourier transforms can be calculated using the fast Fourier transform (FFT) algorithm. This is discussed in §13.9.

Multidimensional integrals are another whole multidimensional bag of worms. Section 4.6 is an introductory discussion in this chapter; the important technique of *Monte-Carlo integration* is treated in Chapter 7.

CITED REFERENCES AND FURTHER READING:

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Chapter 2.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 7.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 3.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 5.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.2, p. 89. [1]

Davis, P., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed. (Orlando, FL: Academic Press).

4.1 Classical Formulas for Equally Spaced Abscissas

Where would any book on numerical analysis be without Mr. Simpson and his "rule"? The classical formulas for integrating a function whose value is known at equally spaced steps have a certain elegance about them, and they are redolent with historical association. Through them, the modern numerical analyst communes with the spirits of his or her predecessors back across the centuries, as far as the time of Newton, if not farther. Alas, times *do* change; with the exception of two of the most modest formulas ("extended trapezoidal rule," equation 4.1.11, and "extended

of various *orders*, with higher order sometimes, but not always, giving higher accuracy. "Romberg integration," which is discussed in §4.3, is a general formalism for making use of integration methods of a variety of different orders, and we recommend it highly.

Apart from the methods of this chapter and of Chapter 16, there are yet other methods for obtaining integrals. One important class is based on function approximation. We discuss explicitly the integration of functions by Chebyshev approximation ("Clenshaw-Curtis" quadrature) in §5.9. Although not explicitly discussed here, you ought to be able to figure out how to do *cubic spline quadrature* using the output of the routine `spline` in §3.3. (Hint: Integrate equation 3.3.3 over x analytically. See [1].)

Some integrals related to Fourier transforms can be calculated using the fast Fourier transform (FFT) algorithm. This is discussed in §13.9.

Multidimensional integrals are another whole multidimensional bag of worms. Section 4.6 is an introductory discussion in this chapter; the important technique of *Monte-Carlo integration* is treated in Chapter 7.

CITED REFERENCES AND FURTHER READING:

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Chapter 2.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 7.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 3.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 5.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.2, p. 89. [1]

Davis, P., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed. (Orlando, FL: Academic Press).

4.1 Classical Formulas for Equally Spaced Abscissas

Where would any book on numerical analysis be without Mr. Simpson and his "rule"? The classical formulas for integrating a function whose value is known at equally spaced steps have a certain elegance about them, and they are redolent with historical association. Through them, the modern numerical analyst communes with the spirits of his or her predecessors back across the centuries, as far as the time of Newton, if not farther. Alas, times *do* change; with the exception of two of the most modest formulas ("extended trapezoidal rule," equation 4.1.11, and "extended

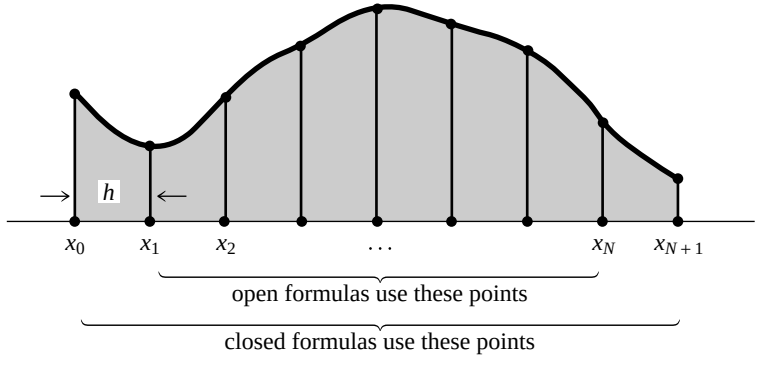


Figure 4.1.1. Quadrature formulas with equally spaced abscissas compute the integral of a function between x_0 and x_{N+1}. Closed formulas evaluate the function on the boundary points, while open formulas refrain from doing so (useful if the evaluation algorithm breaks down on the boundary points).

midpoint rule," equation 4.1.19, see §4.2), the classical formulas are almost entirely useless. They are museum pieces, but beautiful ones.

Some notation: We have a sequence of abscissas, denoted $x_0, x_1, \ldots, x_N$, x_{N+1} which are spaced apart by a constant step h,

$$x_i = x_0 + ih \qquad i = 0, 1, \ldots, N+1 \tag{4.1.1}$$

A function $f(x)$ has known values at the x_i's,

$$f(x_i) \equiv f_i \tag{4.1.2}$$

We want to integrate the function $f(x)$ between a lower limit a and an upper limit b, where a and b are each equal to one or the other of the x_i's. An integration formula that uses the value of the function at the endpoints, $f(a)$ or $f(b)$, is called a *closed* formula. Occasionally, we want to integrate a function whose value at one or both endpoints is difficult to compute (e.g., the computation of f goes to a limit of zero over zero there, or worse yet has an integrable singularity there). In this case we want an *open* formula, which estimates the integral using only x_i's strictly *between* a and b (see Figure 4.1.1).

The basic building blocks of the classical formulas are rules for integrating a function over a small number of intervals. As that number increases, we can find rules that are exact for polynomials of increasingly high order. (Keep in mind that higher order does not always imply higher accuracy in real cases.) A sequence of such closed formulas is now given.

Closed Newton-Cotes Formulas

Trapezoidal rule:

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] \quad + O(h^3 f'') \tag{4.1.3}$$

Here the error term $O(\ )$ signifies that the true answer differs from the estimate by an amount that is the product of some numerical coefficient times h^3 times the value

of the function's second derivative somewhere in the interval of integration. The coefficient is knowable, and it can be found in all the standard references on this subject. The point at which the second derivative is to be evaluated is, however, unknowable. If we knew it, we could evaluate the function there and have a higher-order method! Since the product of a knowable and an unknowable is unknowable, we will streamline our formulas and write only $O(\ )$, instead of the coefficient.

Equation (4.1.3) is a two-point formula (x_1 and x_2). It is exact for polynomials up to and including degree 1, i.e., $f(x) = x$. One anticipates that there is a three-point formula exact up to polynomials of degree 2. This is true; moreover, by a cancellation of coefficients due to left-right symmetry of the formula, the three-point formula is exact for polynomials up to and including degree 3, i.e., $f(x) = x^3$:

Simpson's rule:

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] \quad + O(h^5 f^{(4)}) \tag{4.1.4}$$

Here $f^{(4)}$ means the fourth derivative of the function f evaluated at an unknown place in the interval. Note also that the formula gives the integral over an interval of size $2h$, so the coefficients add up to 2.

There is no lucky cancellation in the four-point formula, so it is also exact for polynomials up to and including degree 3.

Simpson's $\frac{3}{8}$ rule:

$$\int_{x_1}^{x_4} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{9}{8}f_2 + \frac{9}{8}f_3 + \frac{3}{8}f_4\right] \quad + O(h^5 f^{(4)}) \tag{4.1.5}$$

The five-point formula again benefits from a cancellation:

Bode's rule:

$$\int_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f_1 + \frac{64}{45}f_2 + \frac{24}{45}f_3 + \frac{64}{45}f_4 + \frac{14}{45}f_5\right] \quad + O(h^7 f^{(6)}) \tag{4.1.6}$$

This is exact for polynomials up to and including degree 5.

At this point the formulas stop being named after famous personages, so we will not go any further. Consult [1] for additional formulas in the sequence.

Extrapolative Formulas for a Single Interval

We are going to depart from historical practice for a moment. Many texts would give, at this point, a sequence of "Newton-Cotes Formulas of Open Type." Here is an example:

$$\int_{x_0}^{x_5} f(x)dx = h\left[\frac{55}{24}f_1 + \frac{5}{24}f_2 + \frac{5}{24}f_3 + \frac{55}{24}f_4\right] \quad + O(h^5 f^{(4)})$$

Notice that the integral from $a = x_0$ to $b = x_5$ is estimated, using only the interior points x_1, x_2, x_3, x_4. In our opinion, formulas of this type are not useful for the reasons that (i) they cannot usefully be strung together to get "extended" rules, as we are about to do with the closed formulas, and (ii) for all other possible uses they are dominated by the Gaussian integration formulas which we will introduce in §4.5.

Instead of the Newton-Cotes open formulas, let us set out the formulas for estimating the integral in the single interval from x_0 to x_1, using values of the function f at $x_1, x_2, \ldots$. These will be useful building blocks for the "extended" open formulas.

$$\int_{x_0}^{x_1} f(x)dx = h[f_1] \quad + O(h^2 f') \tag{4.1.7}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{3}{2}f_1 - \frac{1}{2}f_2\right] \quad + O(h^3 f'') \tag{4.1.8}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{23}{12}f_1 - \frac{16}{12}f_2 + \frac{5}{12}f_3\right] \quad + O(h^4 f^{(3)}) \tag{4.1.9}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{55}{24}f_1 - \frac{59}{24}f_2 + \frac{37}{24}f_3 - \frac{9}{24}f_4\right] \quad + O(h^5 f^{(4)}) \tag{4.1.10}$$

Perhaps a word here would be in order about how formulas like the above can be derived. There are elegant ways, but the most straightforward is to write down the basic form of the formula, replacing the numerical coefficients with unknowns, say p, q, r, s. Without loss of generality take $x_0 = 0$ and $x_1 = 1$, so $h = 1$. Substitute in turn for $f(x)$ (and for f_1, f_2, f_3, f_4) the functions $f(x) = 1$, $f(x) = x$, $f(x) = x^2$, and $f(x) = x^3$. Doing the integral in each case reduces the left-hand side to a number, and the right-hand side to a linear equation for the unknowns p, q, r, s. Solving the four equations produced in this way gives the coefficients.

Extended Formulas (Closed)

If we use equation (4.1.3) $N - 1$ times, to do the integration in the intervals $(x_1, x_2), (x_2, x_3), \ldots, (x_{N-1}, x_N)$, and then add the results, we obtain an "extended" or "composite" formula for the integral from x_1 to x_N.

Extended trapezoidal rule:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] \quad + O\left(\frac{(b-a)^3 f''}{N^2}\right) \tag{4.1.11}$$

Here we have written the error estimate in terms of the interval $b - a$ and the number of points N instead of in terms of h. This is clearer, since one is usually holding a and b fixed and wanting to know (e.g.) how much the error will be decreased

by taking twice as many steps (in this case, it is by a factor of 4). In subsequent equations we will show *only* the scaling of the error term with the number of steps.

For reasons that will not become clear until §4.2, equation (4.1.11) is in fact the most important equation in this section, the basis for most practical quadrature schemes.

The *extended formula of order* $1/N^3$ is:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{5}{12}f_1 + \frac{13}{12}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.12}$$

(We will see in a moment where this comes from.)

If we apply equation (4.1.4) to successive, nonoverlapping *pairs* of intervals, we get the *extended Simpson's rule:*

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \cdots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.13}$$

Notice that the 2/3, 4/3 alternation continues throughout the interior of the evaluation. Many people believe that the wobbling alternation somehow contains deep information about the integral of their function that is not apparent to mortal eyes. In fact, the alternation is an artifact of using the building block (4.1.4). Another extended formula with the same order as Simpson's rule is

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{7}{6}f_2 + \frac{23}{24}f_3 + f_4 + f_5 + \cdots + f_{N-4} + f_{N-3} + \frac{23}{24}f_{N-2} + \frac{7}{6}f_{N-1} + \frac{3}{8}f_N\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.14}$$

This equation is constructed by fitting cubic polynomials through successive groups of four points; we defer details to §18.3, where a similar technique is used in the solution of integral equations. We can, however, tell you where equation (4.1.12) came from. It is Simpson's extended rule, averaged with a modified version of itself in which the first and last step are done with the trapezoidal rule (4.1.3). The trapezoidal step is *two* orders lower than Simpson's rule; however, its contribution to the integral goes down as an additional power of N (since it is used only twice, not N times). This makes the resulting formula of degree *one* less than Simpson.

Extended Formulas (Open and Semi-open)

We can construct open and semi-open extended formulas by adding the closed formulas (4.1.11)–(4.1.14), evaluated for the second and subsequent steps, to the extrapolative open formulas for the first step, (4.1.7)–(4.1.10). As discussed immediately above, it is consistent to use an end step that is of one order lower than the (repeated) interior step. The resulting formulas for an interval open at both ends are as follows:

Equations (4.1.7) and (4.1.11) give

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{3}{2}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{3}{2}f_{N-1}\right] \quad + O\left(\frac{1}{N^2}\right) \tag{4.1.15}$$

Equations (4.1.8) and (4.1.12) give

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h\Bigg[&\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \\ &\cdots + f_{N-3} + \frac{7}{12}f_{N-2} + \frac{23}{12}f_{N-1}\Bigg] \\ &+ O\left(\frac{1}{N^3}\right)\end{aligned} \tag{4.1.16}$$

Equations (4.1.9) and (4.1.13) give

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h\Bigg[&\frac{27}{12}f_2 + 0 + \frac{13}{12}f_4 + \frac{4}{3}f_5 + \\ &\cdots + \frac{4}{3}f_{N-4} + \frac{13}{12}f_{N-3} + 0 + \frac{27}{12}f_{N-1}\Bigg] \\ &+ O\left(\frac{1}{N^4}\right)\end{aligned} \tag{4.1.17}$$

The interior points alternate 4/3 and 2/3. If we want to avoid this alternation, we can combine equations (4.1.9) and (4.1.14), giving

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h\Bigg[&\frac{55}{24}f_2 - \frac{1}{6}f_3 + \frac{11}{8}f_4 + f_5 + f_6 + f_7 + \\ &\cdots + f_{N-5} + f_{N-4} + \frac{11}{8}f_{N-3} - \frac{1}{6}f_{N-2} + \frac{55}{24}f_{N-1}\Bigg] \\ &+ O\left(\frac{1}{N^4}\right)\end{aligned} \tag{4.1.18}$$

We should mention in passing another extended open formula, for use where the limits of integration are located halfway between tabulated abscissas. This one is known as the *extended midpoint rule*, and is accurate to the same order as (4.1.15):

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h[&f_{3/2} + f_{5/2} + f_{7/2} + \\ &\cdots + f_{N-3/2} + f_{N-1/2}] \quad + O\left(\frac{1}{N^2}\right)\end{aligned} \tag{4.1.19}$$

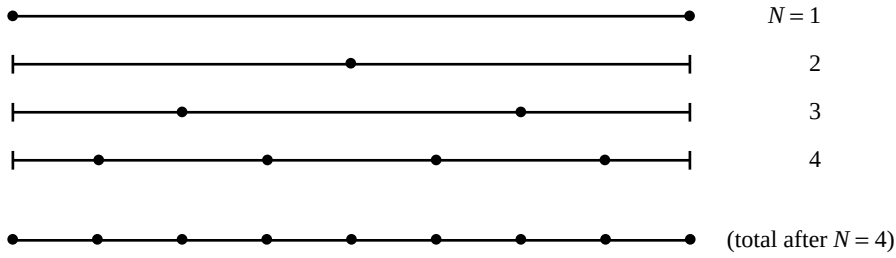


Figure 4.2.1. Sequential calls to the routine `trapzd` incorporate the information from previous calls and evaluate the integrand only at those new points necessary to refine the grid. The bottom line shows the totality of function evaluations after the fourth call. The routine `qsimp`, by weighting the intermediate results, transforms the trapezoid rule into Simpson's rule with essentially no additional overhead.

There are also formulas of higher order for this situation, but we will refrain from giving them.

The *semi-open formulas* are just the obvious combinations of equations (4.1.11)–(4.1.14) with (4.1.15)–(4.1.18), respectively. At the closed end of the integration, use the weights from the former equations; at the open end use the weights from the latter equations. One example should give the idea, the formula with error term decreasing as $1/N^3$ which is closed on the right and open on the left:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.20}$$

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §7.1.

4.2 Elementary Algorithms

Our starting point is equation (4.1.11), the extended trapezoidal rule. There are two facts about the trapezoidal rule which make it the starting point for a variety of algorithms. One fact is rather obvious, while the second is rather "deep."

The obvious fact is that, for a fixed function $f(x)$ to be integrated between fixed limits a and b, one can double the number of intervals in the extended trapezoidal rule without losing the benefit of previous work. The coarsest implementation of the trapezoidal rule is to average the function at its endpoints a and b. The first stage of refinement is to add to this average the value of the function at the halfway point. The second stage of refinement is to add the values at the 1/4 and 3/4 points. And so on (see Figure 4.2.1).

Without further ado we can write a routine with this kind of logic to it:

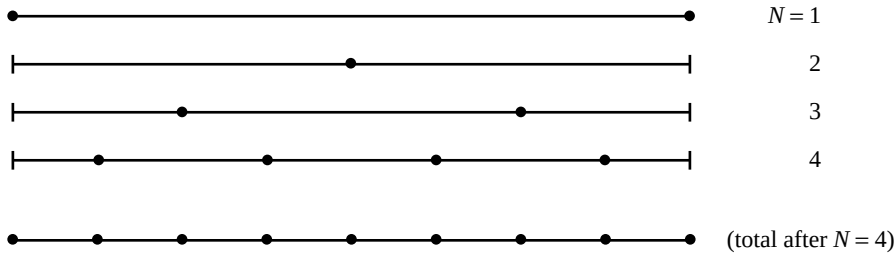


Figure 4.2.1. Sequential calls to the routine `trapzd` incorporate the information from previous calls and evaluate the integrand only at those new points necessary to refine the grid. The bottom line shows the totality of function evaluations after the fourth call. The routine `qsimp`, by weighting the intermediate results, transforms the trapezoid rule into Simpson's rule with essentially no additional overhead.

There are also formulas of higher order for this situation, but we will refrain from giving them.

The *semi-open formulas* are just the obvious combinations of equations (4.1.11)–(4.1.14) with (4.1.15)–(4.1.18), respectively. At the closed end of the integration, use the weights from the former equations; at the open end use the weights from the latter equations. One example should give the idea, the formula with error term decreasing as $1/N^3$ which is closed on the right and open on the left:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.20}$$

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §7.1.

4.2 Elementary Algorithms

Our starting point is equation (4.1.11), the extended trapezoidal rule. There are two facts about the trapezoidal rule which make it the starting point for a variety of algorithms. One fact is rather obvious, while the second is rather "deep."

The obvious fact is that, for a fixed function $f(x)$ to be integrated between fixed limits a and b, one can double the number of intervals in the extended trapezoidal rule without losing the benefit of previous work. The coarsest implementation of the trapezoidal rule is to average the function at its endpoints a and b. The first stage of refinement is to add to this average the value of the function at the halfway point. The second stage of refinement is to add the values at the 1/4 and 3/4 points. And so on (see Figure 4.2.1).

Without further ado we can write a routine with this kind of logic to it:

```
      SUBROUTINE trapzd(func,a,b,s,n)
      INTEGER n
      REAL a,b,s,func
      EXTERNAL func
          This routine computes the nth stage of refinement of an extended trapezoidal rule. func is
          input as the name of the function to be integrated between limits a and b, also input. When
          called with n=1, the routine returns as s the crudest estimate of ∫_a^b f(x)dx. Subsequent
          calls with n=2,3,... (in that sequential order) will improve the accuracy of s by adding 2^(n-2)
          additional interior points. s should not be modified between sequential calls.
      INTEGER it,j
      REAL del,sum,tnm,x
      if (n.eq.1) then
          s=0.5*(b-a)*(func(a)+func(b))
      else
          it=2**(n-2)
          tnm=it
          del=(b-a)/tnm                 This is the spacing of the points to be added.
          x=a+0.5*del
          sum=0.
          do 11 j=1,it
              sum=sum+func(x)
              x=x+del
          enddo 11
          s=0.5*(s+(b-a)*sum/tnm)       This replaces s by its refined value.
      endif
      return
      END
```

The above routine (`trapzd`) is a workhorse that can be harnessed in several ways. The simplest and crudest is to integrate a function by the extended trapezoidal rule where you know in advance (we can't imagine how!) the number of steps you want. If you want $2^M + 1$, you can accomplish this by the fragment

```
      do 11 j=1,m+1
          call trapzd(func,a,b,s,j)
      enddo 11
```

with the answer returned as `s`.

Much better, of course, is to refine the trapezoidal rule until some specified degree of accuracy has been achieved:

```
      SUBROUTINE qtrap(func,a,b,s)
      INTEGER JMAX
      REAL a,b,func,s,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20)
C     USES trapzd
          Returns as s the integral of the function func from a to b. The parameters EPS can be set
          to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
          allowed number of steps. Integration is performed by the trapezoidal rule.
      INTEGER j
      REAL olds
      olds=-1.e30                       Any number that is unlikely to be the average of the function
      do 11 j=1,JMAX                        at its endpoints will do here.
          call trapzd(func,a,b,s,j)
          if (j.gt.5) then              Avoid spurious early convergence.
              if (abs(s-olds).lt.EPS*abs(olds).or.
     *            (s.eq.0..and.olds.eq.0.)) return
          endif
          olds=s
      enddo 11
      pause 'too many steps in qtrap'
      END
```

Unsophisticated as it is, routine `qtrap` is in fact a fairly robust way of doing integrals of functions that are not very smooth. Increased sophistication will usually translate into a higher-order method whose efficiency will be greater only for sufficiently smooth integrands. `qtrap` is the method of choice, e.g., for an integrand which is a function of a variable that is linearly interpolated between measured data points. Be sure that you do not require too stringent an `EPS`, however: If `qtrap` takes too many steps in trying to achieve your required accuracy, accumulated roundoff errors may start increasing, and the routine may never converge. A value 10^{-6} is just on the edge of trouble for most 32-bit machines; it is achievable when the convergence is moderately rapid, but not otherwise.

We come now to the "deep" fact about the extended trapezoidal rule, equation (4.1.11). It is this: The error of the approximation, which begins with a term of order $1/N^2$, is in fact *entirely even* when expressed in powers of $1/N$. This follows directly from the *Euler-Maclaurin Summation Formula*,

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] - \frac{B_2h^2}{2!}(f'_N - f'_1) - \cdots - \frac{B_{2k}h^{2k}}{(2k)!}(f_N^{(2k-1)} - f_1^{(2k-1)}) - \cdots \tag{4.2.1}$$

Here B_{2k} is a *Bernoulli number*, defined by the generating function

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!} \tag{4.2.2}$$

with the first few even values (odd values vanish except for $B_1 = -1/2$)

$$B_0 = 1 \quad B_2 = \frac{1}{6} \quad B_4 = -\frac{1}{30} \quad B_6 = \frac{1}{42} \quad B_8 = -\frac{1}{30} \quad B_{10} = \frac{5}{66} \quad B_{12} = -\frac{691}{2730} \tag{4.2.3}$$

Equation (4.2.1) is not a convergent expansion, but rather only an asymptotic expansion whose error when truncated at any point is always less than twice the magnitude of the first neglected term. The reason that it is not convergent is that the Bernoulli numbers become very large, e.g.,

$$B_{50} = \frac{495057205241079648212477525}{66}$$

The key point is that only even powers of h occur in the error series of (4.2.1). This fact is not, in general, shared by the higher-order quadrature rules in §4.1. For example, equation (4.1.12) has an error series beginning with $O(1/N^3)$, but continuing with all subsequent powers of N: $1/N^4$, $1/N^5$, etc.

Suppose we evaluate (4.1.11) with N steps, getting a result S_N, and then again with $2N$ steps, getting a result S_{2N}. (This is done by any two consecutive calls of

trapzd.) The leading error term in the second evaluation will be 1/4 the size of the error in the first evaluation. Therefore the combination

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \tag{4.2.4}$$

will cancel out the leading order error term. But there *is* no error term of order $1/N^3$, by (4.2.1). The surviving error is of order $1/N^4$, the same as Simpson's rule. In fact, it should not take long for you to see that (4.2.4) is *exactly* Simpson's rule (4.1.13), alternating 2/3's, 4/3's, and all. This is the preferred method for evaluating that rule, and we can write it as a routine exactly analogous to qtrap above:

```
      SUBROUTINE qsimp(func,a,b,s)
      INTEGER JMAX
      REAL a,b,func,s,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20)
C     USES trapzd
        Returns as s the integral of the function func from a to b. The parameters EPS can be set
        to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
        allowed number of steps. Integration is performed by Simpson's rule.
      INTEGER j
      REAL os,ost,st
      ost=-1.e30
      os= -1.e30
      do 11 j=1,JMAX
          call trapzd(func,a,b,st,j)
          s=(4.*st-ost)/3.                  Compare equation (4.2.4), above.
          if (j.gt.5) then                  Avoid spurious early convergence.
              if (abs(s-os).lt.EPS*abs(os).or.
     *            (s.eq.0..and.os.eq.0.)) return
          endif
          os=s
          ost=st
      enddo 11
      pause 'too many steps in qsimp'
      END
```

The routine qsimp will in general be more efficient than qtrap (i.e., require fewer function evaluations) when the function to be integrated has a finite 4th derivative (i.e., a continuous 3rd derivative). The combination of qsimp and its necessary workhorse trapzd is a good one for light-duty work.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.3.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.

4.3 Romberg Integration

We can view Romberg's method as the natural generalization of the routine qsimp in the last section to integration schemes that are of higher order than Simpson's rule. The basic idea is to use the results from k successive refinements of the extended trapezoidal rule (implemented in trapzd) to remove all terms in the error series up to but not including $O(1/N^{2k})$. The routine qsimp is the case of $k = 2$. This is one example of a very general idea that goes by the name of *Richardson's deferred approach to the limit*: Perform some numerical algorithm for various values of a parameter h, and then extrapolate the result to the continuum limit $h = 0$.

Equation (4.2.4), which subtracts off the leading error term, is a special case of polynomial extrapolation. In the more general Romberg case, we can use Neville's algorithm (see §3.1) to extrapolate the successive refinements to zero stepsize. Neville's algorithm can in fact be coded very concisely within a Romberg integration routine. For clarity of the program, however, it seems better to do the extrapolation by subroutine call to polint, already given in §3.1.

```
      SUBROUTINE qromb(func,a,b,ss)
      INTEGER JMAX,JMAXP,K,KM
      REAL a,b,func,ss,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20, JMAXP=JMAX+1, K=5, KM=K-1)
C     USES polint,trapzd
        Returns as ss the integral of the function func from a to b. Integration is performed by
        Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.
        Parameters: EPS is the fractional accuracy desired, as determined by the extrapolation
        error estimate; JMAX limits the total number of steps; K is the number of points used in
        the extrapolation.
      INTEGER j
      REAL dss,h(JMAXP),s(JMAXP)          These store the successive trapezoidal approximations
      h(1)=1.                                 and their relative stepsizes.
      do 11 j=1,JMAX
          call trapzd(func,a,b,s(j),j)
          if (j.ge.K) then
              call polint(h(j-KM),s(j-KM),K,0.,ss,dss)
              if (abs(dss).le.EPS*abs(ss)) return
          endif
          s(j+1)=s(j)
          h(j+1)=0.25*h(j)                This is a key step: The factor is 0.25 even though
      enddo 11                                the stepsize is decreased by only 0.5. This makes
      pause 'too many steps in qromb'         the extrapolation a polynomial in h^2 as allowed
      END                                     by equation (4.2.1), not just a polynomial in h.
```

The routine qromb, along with its required trapzd and polint, is quite powerful for sufficiently smooth (e.g., analytic) integrands, integrated over intervals which contain no singularities, and where the endpoints are also nonsingular. qromb, in such circumstances, takes many, *many* fewer function evaluations than either of the routines in §4.2. For example, the integral

$$\int_0^2 x^4 \log(x + \sqrt{x^2 + 1})dx$$

4.3 Romberg Integration

We can view Romberg's method as the natural generalization of the routine qsimp in the last section to integration schemes that are of higher order than Simpson's rule. The basic idea is to use the results from k successive refinements of the extended trapezoidal rule (implemented in trapzd) to remove all terms in the error series up to but not including $O(1/N^{2k})$. The routine qsimp is the case of $k = 2$. This is one example of a very general idea that goes by the name of *Richardson's deferred approach to the limit*: Perform some numerical algorithm for various values of a parameter h, and then extrapolate the result to the continuum limit $h = 0$.

Equation (4.2.4), which subtracts off the leading error term, is a special case of polynomial extrapolation. In the more general Romberg case, we can use Neville's algorithm (see §3.1) to extrapolate the successive refinements to zero stepsize. Neville's algorithm can in fact be coded very concisely within a Romberg integration routine. For clarity of the program, however, it seems better to do the extrapolation by subroutine call to polint, already given in §3.1.

```
      SUBROUTINE qromb(func,a,b,ss)
      INTEGER JMAX,JMAXP,K,KM
      REAL a,b,func,ss,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20, JMAXP=JMAX+1, K=5, KM=K-1)
C     USES polint,trapzd
        Returns as ss the integral of the function func from a to b. Integration is performed by
        Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.
        Parameters: EPS is the fractional accuracy desired, as determined by the extrapolation
        error estimate; JMAX limits the total number of steps; K is the number of points used in
        the extrapolation.
      INTEGER j
      REAL dss,h(JMAXP),s(JMAXP)          These store the successive trapezoidal approximations
      h(1)=1.                                 and their relative stepsizes.
      do 11 j=1,JMAX
          call trapzd(func,a,b,s(j),j)
          if (j.ge.K) then
              call polint(h(j-KM),s(j-KM),K,0.,ss,dss)
              if (abs(dss).le.EPS*abs(ss)) return
          endif
          s(j+1)=s(j)
          h(j+1)=0.25*h(j)                This is a key step: The factor is 0.25 even though
      enddo 11                                the stepsize is decreased by only 0.5. This makes
      pause 'too many steps in qromb'         the extrapolation a polynomial in h^2 as allowed
      END                                     by equation (4.2.1), not just a polynomial in h.
```

The routine qromb, along with its required trapzd and polint, is quite powerful for sufficiently smooth (e.g., analytic) integrands, integrated over intervals which contain no singularities, and where the endpoints are also nonsingular. qromb, in such circumstances, takes many, *many* fewer function evaluations than either of the routines in §4.2. For example, the integral

$$\int_0^2 x^4 \log(x + \sqrt{x^2 + 1})dx$$

converges (with parameters as shown above) on the very first extrapolation, after just 5 calls to `trapzd`, while `qsimp` requires 8 calls (8 times as many evaluations of the integrand) and `qtrap` requires 13 calls (making 256 times as many evaluations of the integrand).

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§3.4–3.5.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §4.10–2.

4.4 Improper Integrals

For our present purposes, an integral will be "improper" if it has any of the following problems:

- its integrand goes to a finite limiting value at finite upper and lower limits, but cannot be evaluated *right on* one of those limits (e.g., $\sin x/x$ at $x = 0$)
- its upper limit is ∞ , or its lower limit is $-\infty$
- it has an integrable singularity at either limit (e.g., $x^{-1/2}$ at $x = 0$)
- it has an integrable singularity at a known place between its upper and lower limits
- it has an integrable singularity at an unknown place between its upper and lower limits

If an integral is infinite (e.g., $\int_1^\infty x^{-1}dx$), or does not exist in a limiting sense (e.g., $\int_{-\infty}^{\infty} \cos x dx$), we do not call it improper; we call it impossible. No amount of clever algorithmics will return a meaningful answer to an ill-posed problem.

In this section we will generalize the techniques of the preceding two sections to cover the first four problems on the above list. A more advanced discussion of quadrature with integrable singularities occurs in Chapter 18, notably §18.3. The fifth problem, singularity at unknown location, can really only be handled by the use of a variable stepsize differential equation integration routine, as will be given in Chapter 16.

We need a workhorse like the extended trapezoidal rule (equation 4.1.11), but one which is an *open* formula in the sense of §4.1, i.e., does not require the integrand to be evaluated at the endpoints. Equation (4.1.19), the extended midpoint rule, is the best choice. The reason is that (4.1.19) shares with (4.1.11) the "deep" property of having an error series that is entirely even in h. Indeed there is a formula, not as well known as it ought to be, called the *Second Euler-Maclaurin summation formula*,

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h[f_{3/2} + f_{5/2} + f_{7/2} + \cdots + f_{N-3/2} + f_{N-1/2}] \\ + \frac{B_2 h^2}{4}(f'_N - f'_1) + \cdots \\ + \frac{B_{2k} h^{2k}}{(2k)!}(1 - 2^{-2k+1})(f_N^{(2k-1)} - f_1^{(2k-1)}) + \cdots\end{aligned} \tag{4.4.1}$$

converges (with parameters as shown above) on the very first extrapolation, after just 5 calls to `trapzd`, while `qsimp` requires 8 calls (8 times as many evaluations of the integrand) and `qtrap` requires 13 calls (making 256 times as many evaluations of the integrand).

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§3.4–3.5.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §4.10–2.

4.4 Improper Integrals

For our present purposes, an integral will be "improper" if it has any of the following problems:

- its integrand goes to a finite limiting value at finite upper and lower limits, but cannot be evaluated *right on* one of those limits (e.g., $\sin x/x$ at $x = 0$)
- its upper limit is ∞ , or its lower limit is $-\infty$
- it has an integrable singularity at either limit (e.g., $x^{-1/2}$ at $x = 0$)
- it has an integrable singularity at a known place between its upper and lower limits
- it has an integrable singularity at an unknown place between its upper and lower limits

If an integral is infinite (e.g., $\int_1^\infty x^{-1}dx$), or does not exist in a limiting sense (e.g., $\int_{-\infty}^{\infty} \cos x dx$), we do not call it improper; we call it impossible. No amount of clever algorithmics will return a meaningful answer to an ill-posed problem.

In this section we will generalize the techniques of the preceding two sections to cover the first four problems on the above list. A more advanced discussion of quadrature with integrable singularities occurs in Chapter 18, notably §18.3. The fifth problem, singularity at unknown location, can really only be handled by the use of a variable stepsize differential equation integration routine, as will be given in Chapter 16.

We need a workhorse like the extended trapezoidal rule (equation 4.1.11), but one which is an *open* formula in the sense of §4.1, i.e., does not require the integrand to be evaluated at the endpoints. Equation (4.1.19), the extended midpoint rule, is the best choice. The reason is that (4.1.19) shares with (4.1.11) the "deep" property of having an error series that is entirely even in h. Indeed there is a formula, not as well known as it ought to be, called the *Second Euler-Maclaurin summation formula*,

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h[f_{3/2} + f_{5/2} + f_{7/2} + \cdots + f_{N-3/2} + f_{N-1/2}] \\ + \frac{B_2 h^2}{4}(f'_N - f'_1) + \cdots \\ + \frac{B_{2k} h^{2k}}{(2k)!}(1 - 2^{-2k+1})(f_N^{(2k-1)} - f_1^{(2k-1)}) + \cdots\end{aligned} \tag{4.4.1}$$

This equation can be derived by writing out (4.2.1) with stepsize h, then writing it out again with stepsize $h/2$, then subtracting the first from twice the second.

It is not possible to double the number of steps in the extended midpoint rule and still have the benefit of previous function evaluations (try it!). However, it is possible to *triple* the number of steps and do so. Shall we do this, or double and accept the loss? On the average, tripling does a factor $\sqrt{3}$ of unnecessary work, since the "right" number of steps for a desired accuracy criterion may in fact fall anywhere in the logarithmic interval implied by tripling. For doubling, the factor is only $\sqrt{2}$, but we lose an extra factor of 2 in being unable to use all the previous evaluations. Since $1.732 < 2 \times 1.414$, it is better to triple.

Here is the resulting routine, which is directly comparable to `trapzd`.

```
SUBROUTINE midpnt(func,a,b,s,n)
INTEGER n
REAL a,b,s,func
EXTERNAL func
    This routine computes the nth stage of refinement of an extended midpoint rule. func is
    input as the name of the function to be integrated between limits a and b, also input. When
    called with n=1, the routine returns as s the crudest estimate of ∫_a^b f(x)dx. Subsequent
    calls with n=2,3,... (in that sequential order) will improve the accuracy of s by adding
    (2/3) × 3^(n-1) additional interior points. s should not be modified between sequential calls.
INTEGER it,j
REAL ddel,del,sum,tnm,x
if (n.eq.1) then
    s=(b-a)*func(0.5*(a+b))
else
    it=3**(n-2)
    tnm=it
    del=(b-a)/(3.*tnm)
    ddel=del+del                The added points alternate in spacing between del and ddel.
    x=a+0.5*del
    sum=0.
    do 11 j=1,it
        sum=sum+func(x)
        x=x+ddel
        sum=sum+func(x)
        x=x+del
    enddo 11
    s=(s+(b-a)*sum/tnm)/3.      The new sum is combined with the old integral to give a
endif                               refined integral.
return
END
```

The routine `midpnt` can exactly replace `trapzd` in a driver routine like `qtrap` (§4.2); one simply changes `call trapzd` to `call midpnt`, and perhaps also decreases the parameter `JMAX` since $3^{\mathtt{JMAX}-1}$ (from step tripling) is a much larger number than $2^{\mathtt{JMAX}-1}$ (step doubling).

The open formula implementation analogous to Simpson's rule (`qsimp` in §4.2) substitutes `midpnt` for `trapzd` and decreases `JMAX` as above, but now also changes the extrapolation step to be

```
    s=(9.*st-ost)/8.
```

since, when the number of steps is tripled, the error decreases to $1/9$th its size, not $1/4$th as with step doubling.

Either the modified qtrap or the modified qsimp will fix the first problem on the list at the beginning of this section. Yet more sophisticated is to generalize Romberg integration in like manner:

```
      SUBROUTINE qromo(func,a,b,ss,choose)
      INTEGER JMAX,JMAXP,K,KM
      REAL a,b,func,ss,EPS
      EXTERNAL func,choose
      PARAMETER (EPS=1.e-6, JMAX=14, JMAXP=JMAX+1, K=5, KM=K-1)
C     USES polint
```

Romberg integration on an open interval. Returns as ss the integral of the function func from a to b, using any specified integrating subroutine choose and Romberg's method. Normally choose will be an open formula, not evaluating the function at the endpoints. It is assumed that choose triples the number of steps on each call, and that its error series contains only even powers of the number of steps. The routines midpnt, midinf, midsql, midsqu, are possible choices for choose. The parameters have the same meaning as in qromb.

```
      INTEGER j
      REAL dss,h(JMAXP),s(JMAXP)
      h(1)=1.
      do 11 j=1,JMAX
          call choose(func,a,b,s(j),j)
          if (j.ge.K) then
              call polint(h(j-KM),s(j-KM),K,0.,ss,dss)
              if (abs(dss).le.EPS*abs(ss)) return
          endif
          s(j+1)=s(j)
          h(j+1)=h(j)/9.
      enddo 11
      pause 'too many steps in qromo'
      END
```

This is where the assumption of step tripling and an even error series is used.

The differences between qromo and qromb (§4.3) are so slight that it is perhaps gratuitous to list qromo in full. It, however, is an excellent driver routine for solving all the other problems of improper integrals in our first list (except the intractable fifth), as we shall now see.

The basic trick for improper integrals is to make a change of variables to eliminate the singularity, or to map an infinite range of integration to a finite one. For example, the identity

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt \qquad ab > 0 \tag{4.4.2}$$

can be used with *either* $b \to \infty$ and a positive, *or* with $a \to -\infty$ and b negative, and works for any function which decreases towards infinity faster than $1/x^2$.

You can make the change of variable implied by (4.4.2) either analytically and then use (e.g.) qromo and midpnt to do the numerical evaluation, *or* you can let the numerical algorithm make the change of variable for you. We prefer the latter method as being more transparent to the user. To implement equation (4.4.2) we simply write a modified version of midpnt, called midinf, which allows b to be infinite (or, more precisely, a very large number on your particular machine, such as 1×10^{30}), or a to be negative and infinite.

```
SUBROUTINE midinf(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement
    of the integral of funk from aa to bb, except that the function is evaluated at evenly spaced
    points in 1/x rather than in x. This allows the upper limit bb to be as large and positive
    as the computer allows, or the lower limit aa to be as large and negative, but not both.
    aa and bb must have the same sign.
INTEGER it,j
REAL a,b,ddel,del,sum,tnm,func,x
func(x)=funk(1./x)/x**2          This statement function effects the change of variable.
b=1./aa                          These two statements change the limits of integration ac-
a=1./bb                              cordingly.
if (n.eq.1) then                 From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*func(0.5*(a+b))
else
    it=3**(n-2)
    tnm=it
    del=(b-a)/(3.*tnm)
    ddel=del+del
    x=a+0.5*del
    sum=0.
    do 11 j=1,it
        sum=sum+func(x)
        x=x+ddel
        sum=sum+func(x)
        x=x+del
    enddo 11
    s=(s+(b-a)*sum/tnm)/3.
endif
return
END
```

If you need to integrate from a negative lower limit to positive infinity, you do this by breaking the integral into two pieces at some positive value, for example,

```
call qromo(funk,-5.,2.,s1,midpnt)
call qromo(funk,2.,1.e30,s2,midinf)
answer=s1+s2
```

Where should you choose the breakpoint? At a sufficiently large positive value so that the function funk is at least beginning to approach its asymptotic decrease to zero value at infinity. The polynomial extrapolation implicit in the second call to qromo deals with a polynomial in $1/x$, not in x.

To deal with an integral that has an integrable power-law singularity at its lower limit, one also makes a change of variable. If the integrand diverges as $(x-a)^\gamma$, $0 \le \gamma < 1$, near $x = a$, use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(t^{\frac{1}{1-\gamma}} + a)dt \qquad (b > a) \qquad (4.4.3)$$

If the singularity is at the upper limit, use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(b - t^{\frac{1}{1-\gamma}})dt \qquad (b > a) \qquad (4.4.4)$$

If there is a singularity at both limits, divide the integral at an interior breakpoint as in the example above.

Equations (4.4.3) and (4.4.4) are particularly simple in the case of inverse square-root singularities, a case that occurs frequently in practice:

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(a+t^2)dt \qquad (b>a) \tag{4.4.5}$$

for a singularity at a, and

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(b-t^2)dt \qquad (b>a) \tag{4.4.6}$$

for a singularity at b. Once again, we can implement these changes of variable transparently to the user by defining substitute routines for `midpnt` which make the change of variable automatically:

```
SUBROUTINE midsql(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, except that it allows for an inverse square-
    root singularity in the integrand at the lower limit aa.
INTEGER it,j
REAL ddel,del,sum,tnm,x,func,a,b
func(x)=2.*x*funk(aa+x**2)
b=sqrt(bb-aa)
a=0.
if (n.eq.1) then
    The rest of the routine is exactly like midpnt and is omitted.
```

Similarly,

```
SUBROUTINE midsqu(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, except that it allows for an inverse square-
    root singularity in the integrand at the upper limit bb.
INTEGER it,j
REAL ddel,del,sum,tnm,x,func,a,b
func(x)=2.*x*funk(bb-x**2)
b=sqrt(bb-aa)
a=0.
if (n.eq.1) then
    The rest of the routine is exactly like midpnt and is omitted.
```

One last example should suffice to show how these formulas are derived in general. Suppose the upper limit of integration is infinite, and the integrand falls off exponentially. Then we want a change of variable that maps $e^{-x}dx$ into $(\pm)dt$ (with the sign chosen to keep the upper limit of the new variable larger than the lower limit). Doing the integration gives by inspection

$$t = e^{-x} \qquad \text{or} \qquad x = -\log t \tag{4.4.7}$$

so that

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t)\frac{dt}{t} \tag{4.4.8}$$

The user-transparent implementation would be

```
SUBROUTINE midexp(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, except that bb is assumed to be infinite
    (value passed not actually used). It is assumed that the function funk decreases exponen-
    tially rapidly at infinity.
INTEGER it,j
REAL ddel,del,sum,tnm,x,func,a,b
func(x)=funk(-log(x))/x
b=exp(-aa)
a=0.
if (n.eq.1) then
    The rest of the routine is exactly like midpnt and is omitted.
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.3, p. 294.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.7, p. 152.

4.5 Gaussian Quadratures and Orthogonal Polynomials

In the formulas of §4.1, the integral of a function was approximated by the sum of its functional values at a set of equally spaced points, multiplied by certain aptly chosen weighting coefficients. We saw that as we allowed ourselves more freedom in choosing the coefficients, we could achieve integration formulas of higher and higher order. The idea of *Gaussian quadratures* is to give ourselves the freedom to choose not only the weighting coefficients, but also the location of the abscissas at which the function is to be evaluated: They will no longer be equally spaced. Thus, we will have *twice* the number of degrees of freedom at our disposal; it will turn out that we can achieve Gaussian quadrature formulas whose order is, essentially, twice that of the Newton-Cotes formula with the same number of function evaluations.

Does this sound too good to be true? Well, in a sense it is. The catch is a familiar one, which cannot be overemphasized: High order is not the same as high accuracy. High order translates to high accuracy only when the integrand is very smooth, in the sense of being "well-approximated by a polynomial."

so that

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t)\frac{dt}{t} \tag{4.4.8}$$

The user-transparent implementation would be

```
SUBROUTINE midexp(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, except that bb is assumed to be infinite
    (value passed not actually used). It is assumed that the function funk decreases exponen-
    tially rapidly at infinity.
INTEGER it,j
REAL ddel,del,sum,tnm,x,func,a,b
func(x)=funk(-log(x))/x
b=exp(-aa)
a=0.
if (n.eq.1) then
    The rest of the routine is exactly like midpnt and is omitted.
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.3, p. 294.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.7, p. 152.

4.5 Gaussian Quadratures and Orthogonal Polynomials

In the formulas of §4.1, the integral of a function was approximated by the sum of its functional values at a set of equally spaced points, multiplied by certain aptly chosen weighting coefficients. We saw that as we allowed ourselves more freedom in choosing the coefficients, we could achieve integration formulas of higher and higher order. The idea of *Gaussian quadratures* is to give ourselves the freedom to choose not only the weighting coefficients, but also the location of the abscissas at which the function is to be evaluated: They will no longer be equally spaced. Thus, we will have *twice* the number of degrees of freedom at our disposal; it will turn out that we can achieve Gaussian quadrature formulas whose order is, essentially, twice that of the Newton-Cotes formula with the same number of function evaluations.

Does this sound too good to be true? Well, in a sense it is. The catch is a familiar one, which cannot be overemphasized: High order is not the same as high accuracy. High order translates to high accuracy only when the integrand is very smooth, in the sense of being "well-approximated by a polynomial."

There is, however, one additional feature of Gaussian quadrature formulas that adds to their usefulness: We can arrange the choice of weights and abscissas to make the integral exact for a class of integrands "polynomials times some known function $W(x)$" rather than for the usual class of integrands "polynomials." The function $W(x)$ can then be chosen to remove integrable singularities from the desired integral. Given $W(x)$, in other words, and given an integer N, we can find a set of weights w_j and abscissas x_j such that the approximation

$$\int_a^b W(x)f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.1}$$

is exact if $f(x)$ is a polynomial. For example, to do the integral

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}}dx \tag{4.5.2}$$

(not a very natural looking integral, it must be admitted), we might well be interested in a Gaussian quadrature formula based on the choice

$$W(x) = \frac{1}{\sqrt{1-x^2}} \tag{4.5.3}$$

in the interval $(-1, 1)$. (This particular choice is called *Gauss-Chebyshev integration*, for reasons that will become clear shortly.)

Notice that the integration formula (4.5.1) can also be written with the weight function $W(x)$ not overtly visible: Define $g(x) \equiv W(x)f(x)$ and $v_j \equiv w_j/W(x_j)$. Then (4.5.1) becomes

$$\int_a^b g(x)dx \approx \sum_{j=1}^{N} v_j g(x_j) \tag{4.5.4}$$

Where did the function $W(x)$ go? It is lurking there, ready to give high-order accuracy to integrands of the form polynomials times $W(x)$, and ready to *deny* high-order accuracy to integrands that are otherwise perfectly smooth and well-behaved. When you find tabulations of the weights and abscissas for a given $W(x)$, you have to determine carefully whether they are to be used with a formula in the form of (4.5.1), or like (4.5.4).

Here is an example of a quadrature routine that contains the tabulated abscissas and weights for the case $W(x) = 1$ and $N = 10$. Since the weights and abscissas are, in this case, symmetric around the midpoint of the range of integration, there are actually only five distinct values of each:

```
SUBROUTINE qgaus(func,a,b,ss)
REAL a,b,ss,func
EXTERNAL func
    Returns as ss the integral of the function func between a and b, by ten-point Gauss-
    Legendre integration: the function is evaluated exactly ten times at interior points in the
    range of integration.
INTEGER j
```

```
REAL dx,xm,xr,w(5),x(5)        The abscissas and weights.
SAVE w,x
DATA w/.2955242247,.2692667193,.2190863625,.1494513491,.0666713443/
DATA x/.1488743389,.4333953941,.6794095682,.8650633666,.9739065285/
xm=0.5*(b+a)
xr=0.5*(b-a)
ss=0                           Will be twice the average value of the function, since the ten
do 11 j=1,5                        weights (five numbers above each used twice) sum to 2.
    dx=xr*x(j)
    ss=ss+w(j)*(func(xm+dx)+func(xm-dx))
enddo 11
ss=xr*ss                       Scale the answer to the range of integration.
return
END
```

The above routine illustrates that one can use Gaussian quadratures without necessarily understanding the theory behind them: One just locates tabulated weights and abscissas in a book (e.g., [1] or [2]). However, the theory is very pretty, and it will come in handy if you ever need to construct your own tabulation of weights and abscissas for an unusual choice of $W(x)$. We will therefore give, without any proofs, some useful results that will enable you to do this. Several of the results assume that $W(x)$ does not change sign inside (a, b), which is usually the case in practice.

The theory behind Gaussian quadratures goes back to Gauss in 1814, who used continued fractions to develop the subject. In 1826 Jacobi rederived Gauss's results by means of orthogonal polynomials. The systematic treatment of arbitrary weight functions $W(x)$ using orthogonal polynomials is largely due to Christoffel in 1877. To introduce these orthogonal polynomials, let us fix the interval of interest to be (a, b). We can define the "scalar product of two functions f and g over a weight function W" as

$$\langle f|g\rangle \equiv \int_a^b W(x)f(x)g(x)dx \tag{4.5.5}$$

The scalar product is a number, not a function of x. Two functions are said to be *orthogonal* if their scalar product is zero. A function is said to be *normalized* if its scalar product with itself is unity. A set of functions that are all mutually orthogonal and also all individually normalized is called an *orthonormal* set.

We can find a set of polynomials (i) that includes exactly one polynomial of order j, called $p_j(x)$, for each $j = 0, 1, 2, \ldots$, and (ii) all of which are mutually orthogonal over the specified weight function $W(x)$. A constructive procedure for finding such a set is the recurrence relation

$$\begin{aligned} p_{-1}(x) &\equiv 0 \\ p_0(x) &\equiv 1 \\ p_{j+1}(x) &= (x - a_j)p_j(x) - b_j p_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.5.6}$$

where

$$\begin{aligned} a_j &= \frac{\langle xp_j|p_j\rangle}{\langle p_j|p_j\rangle} \qquad j = 0, 1, \ldots \\ b_j &= \frac{\langle p_j|p_j\rangle}{\langle p_{j-1}|p_{j-1}\rangle} \qquad j = 1, 2, \ldots \end{aligned} \tag{4.5.7}$$

The coefficient b_0 is arbitrary; we can take it to be zero.

The polynomials defined by (4.5.6) are *monic*, i.e., the coefficient of their leading term [x^j for $p_j(x)$] is unity. If we divide each $p_j(x)$ by the constant $[\langle p_j|p_j\rangle]^{1/2}$ we can render the set of polynomials orthonormal. One also encounters orthogonal polynomials with various other normalizations. You can convert from a given normalization to monic polynomials if you know that the coefficient of x^j in p_j is λ_j, say; then the monic polynomials are obtained by dividing each p_j by λ_j. Note that the coefficients in the recurrence relation (4.5.6) depend on the adopted normalization.

The polynomial $p_j(x)$ can be shown to have exactly j distinct roots in the interval (a, b). Moreover, it can be shown that the roots of $p_j(x)$ "interleave" the $j-1$ roots of $p_{j-1}(x)$, i.e., there is exactly one root of the former in between each two adjacent roots of the latter. This fact comes in handy if you need to find all the roots: You can start with the one root of $p_1(x)$ and then, in turn, bracket the roots of each higher j, pinning them down at each stage more precisely by Newton's rule or some other root-finding scheme (see Chapter 9).

Why would you ever want to find all the roots of an orthogonal polynomial $p_j(x)$? Because the abscissas of the N-point Gaussian quadrature formulas (4.5.1) and (4.5.4) with weighting function $W(x)$ in the interval (a, b) are precisely the roots of the orthogonal polynomial $p_N(x)$ for the same interval and weighting function. This is the fundamental theorem of Gaussian quadratures, and lets you find the abscissas for any particular case.

Once you know the abscissas $x_1, \ldots, x_N$, you need to find the weights w_j, $j = 1, \ldots, N$. One way to do this (not the most efficient) is to solve the set of linear equations

$$\begin{bmatrix} p_0(x_1) & \ldots & p_0(x_N) \\ p_1(x_1) & \ldots & p_1(x_N) \\ \vdots & & \vdots \\ p_{N-1}(x_1) & \ldots & p_{N-1}(x_N) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} \int_a^b W(x)p_0(x)dx \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.5.8}$$

Equation (4.5.8) simply solves for those weights such that the quadrature (4.5.1) gives the correct answer for the integral of the first N orthogonal polynomials. Note that the zeros on the right-hand side of (4.5.8) appear because $p_1(x), \ldots, p_{N-1}(x)$ are all orthogonal to $p_0(x)$, which is a constant. It can be shown that, with those weights, the integral of the *next* $N-1$ polynomials is also exact, so that the quadrature is exact for all polynomials of degree $2N-1$ or less. Another way to evaluate the weights (though one whose proof is beyond our scope) is by the formula

$$w_j = \frac{\langle p_{N-1}|p_{N-1}\rangle}{p_{N-1}(x_j)p'_N(x_j)} \tag{4.5.9}$$

where $p'_N(x_j)$ is the derivative of the orthogonal polynomial at its zero x_j.

The computation of Gaussian quadrature rules thus involves two distinct phases: (i) the generation of the orthogonal polynomials $p_0, \ldots, p_N$, i.e., the computation of the coefficients a_j, b_j in (4.5.6); (ii) the determination of the zeros of $p_N(x)$, and the computation of the associated weights. For the case of the "classical" orthogonal polynomials, the coefficients a_j and b_j are explicitly known (equations 4.5.10 –

4.5.14 below) and phase (i) can be omitted. However, if you are confronted with a "nonclassical" weight function $W(x)$, and you don't know the coefficients a_j and b_j, the construction of the associated set of orthogonal polynomials is not trivial. We discuss it at the end of this section.

Computation of the Abscissas and Weights

This task can range from easy to difficult, depending on how much you already know about your weight function and its associated polynomials. In the case of classical, well-studied, orthogonal polynomials, practically everything is known, including good approximations for their zeros. These can be used as starting guesses, enabling Newton's method (to be discussed in §9.4) to converge very rapidly. Newton's method requires the derivative $p'_N(x)$, which is evaluated by standard relations in terms of p_N and p_{N-1}. The weights are then conveniently evaluated by equation (4.5.9). For the following named cases, this direct root-finding is faster, by a factor of 3 to 5, than any other method.

Here are the weight functions, intervals, and recurrence relations that generate the most commonly used orthogonal polynomials and their corresponding Gaussian quadrature formulas.

Gauss-Legendre:

$$W(x) = 1 \qquad -1 < x < 1$$

$$(j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1} \tag{4.5.10}$$

Gauss-Chebyshev:

$$W(x) = (1-x^2)^{-1/2} \qquad -1 < x < 1$$

$$T_{j+1} = 2xT_j - T_{j-1} \tag{4.5.11}$$

Gauss-Laguerre:

$$W(x) = x^\alpha e^{-x} \qquad 0 < x < \infty$$

$$(j+1)L^\alpha_{j+1} = (-x+2j+\alpha+1)L^\alpha_j - (j+\alpha)L^\alpha_{j-1} \tag{4.5.12}$$

Gauss-Hermite:

$$W(x) = e^{-x^2} \qquad -\infty < x < \infty$$

$$H_{j+1} = 2xH_j - 2jH_{j-1} \tag{4.5.13}$$

Gauss-Jacobi:

$$W(x) = (1-x)^\alpha(1+x)^\beta \qquad -1 < x < 1$$

$$c_j P_{j+1}^{(\alpha,\beta)} = (d_j + e_j x) P_j^{(\alpha,\beta)} - f_j P_{j-1}^{(\alpha,\beta)} \tag{4.5.14}$$

where the coefficients c_j, d_j, e_j, and f_j are given by

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \tag{4.5.15}$$

We now give individual routines that calculate the abscissas and weights for these cases. First comes the most common set of abscissas and weights, those of Gauss-Legendre. The routine, due to G.B. Rybicki, uses equation (4.5.9) in the special form for the Gauss-Legendre case,

$$w_j = \frac{2}{(1-x_j^2)[P_N'(x_j)]^2} \tag{4.5.16}$$

The routine also scales the range of integration from (x_1, x_2) to $(-1, 1)$, and provides abscissas x_j and weights w_j for the Gaussian formula

$$\int_{x_1}^{x_2} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.17}$$

```
      SUBROUTINE gauleg(x1,x2,x,w,n)
      INTEGER n
      REAL x1,x2,x(n),w(n)
      DOUBLE PRECISION EPS
      PARAMETER (EPS=3.d-14)                    EPS is the relative precision.
        Given the lower and upper limits of integration x1 and x2, and given n, this routine returns
        arrays x(1:n) and w(1:n) of length n, containing the abscissas and weights of the Gauss-
        Legendre n-point quadrature formula.
      INTEGER i,j,m
      DOUBLE PRECISION p1,p2,p3,pp,xl,xm,z,z1
        High precision is a good idea for this routine.
      m=(n+1)/2                                 The roots are symmetric in the interval, so we
      xm=0.5d0*(x2+x1)                            only have to find half of them.
      xl=0.5d0*(x2-x1)
      do 12 i=1,m                               Loop over the desired roots.
        z=cos(3.141592654d0*(i-.25d0)/(n+.5d0))
          Starting with the above approximation to the ith root, we enter the main loop of re-
          finement by Newton's method.
1       continue
          p1=1.d0
          p2=0.d0
          do 11 j=1,n                           Loop up the recurrence relation to get the Leg-
            p3=p2                                 endre polynomial evaluated at z.
            p2=p1
            p1=((2.d0*j-1.d0)*z*p2-(j-1.d0)*p3)/j
          enddo 11
            p1 is now the desired Legendre polynomial. We next compute pp, its derivative, by
            a standard relation involving also p2, the polynomial of one lower order.
          pp=n*(z*p1-p2)/(z*z-1.d0)
```

```
            z1=z
            z=z1-p1/pp                          Newton's method.
        if(abs(z-z1).gt.EPS)goto 1
        x(i)=xm-xl*z                            Scale the root to the desired interval,
        x(n+1-i)=xm+xl*z                        and put in its symmetric counterpart.
        w(i)=2.d0*xl/((1.d0-z*z)*pp*pp)         Compute the weight
        w(n+1-i)=w(i)                           and its symmetric counterpart.
    enddo 12
    return
    END
```

Next we give three routines that use initial approximations for the roots given by Stroud and Secrest [2]. The first is for Gauss-Laguerre abscissas and weights, to be used with the integration formula

$$\int_0^\infty x^\alpha e^{-x} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.18}$$

```
    SUBROUTINE gaulag(x,w,n,alf)
    INTEGER n,MAXIT
    REAL alf,w(n),x(n)
    DOUBLE PRECISION EPS
    PARAMETER (EPS=3.D-14,MAXIT=10)     Increase EPS if you don't have this precision.
C   USES gammln
      Given alf, the parameter α of the Laguerre polynomials, this routine returns arrays x(1:n)
      and w(1:n) containing the abscissas and weights of the n-point Gauss-Laguerre quadrature
      formula. The smallest abscissa is returned in x(1), the largest in x(n).
    INTEGER i,its,j
    REAL ai,gammln
    DOUBLE PRECISION p1,p2,p3,pp,z,z1
      High precision is a good idea for this routine.
    do 13 i=1,n                         Loop over the desired roots.
        if(i.eq.1)then                  Initial guess for the smallest root.
            z=(1.+alf)*(3.+.92*alf)/(1.+2.4*n+1.8*alf)
        else if(i.eq.2)then             Initial guess for the second root.
            z=z+(15.+6.25*alf)/(1.+.9*alf+2.5*n)
        else                            Initial guess for the other roots.
            ai=i-2
            z=z+((1.+2.55*ai)/(1.9*ai)+1.26*ai*alf/
*               (1.+3.5*ai))*(z-x(i-2))/(1.+.3*alf)
        endif
        do 12 its=1,MAXIT               Refinement by Newton's method.
            p1=1.d0
            p2=0.d0
            do 11 j=1,n                 Loop up the recurrence relation to get the Laguerre
                p3=p2                       polynomial evaluated at z.
                p2=p1
                p1=((2*j-1+alf-z)*p2-(j-1+alf)*p3)/j
            enddo 11
              p1 is now the desired Laguerre polynomial. We next compute pp, its derivative, by
              a standard relation involving also p2, the polynomial of one lower order.
            pp=(n*p1-(n+alf)*p2)/z
            z1=z
            z=z1-p1/pp                  Newton's formula.
            if(abs(z-z1).le.EPS)goto 1
        enddo 12
        pause 'too many iterations in gaulag'
1       x(i)=z                          Store the root and the weight.
```

```
    w(i)=-exp(gammln(alf+n)-gammln(float(n)))/(pp*n*p2)
enddo 13
return
END
```

Next is a routine for Gauss-Hermite abscissas and weights. If we use the "standard" normalization of these functions, as given in equation (4.5.13), we find that the computations overflow for large N because of various factorials that occur. We can avoid this by using instead the orthonormal set of polynomials $\widetilde{H}_j$. They are generated by the recurrence

$$\widetilde{H}_{-1} = 0, \quad \widetilde{H}_0 = \frac{1}{\pi^{1/4}}, \quad \widetilde{H}_{j+1} = x\sqrt{\frac{2}{j+1}}\widetilde{H}_j - \sqrt{\frac{j}{j+1}}\widetilde{H}_{j-1} \tag{4.5.19}$$

The formula for the weights becomes

$$w_j = \frac{2}{(\widetilde{H}'_j)^2} \tag{4.5.20}$$

while the formula for the derivative with this normalization is

$$\widetilde{H}'_j = \sqrt{2j}\widetilde{H}_{j-1} \tag{4.5.21}$$

The abscissas and weights returned by `gauher` are used with the integration formula

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.22}$$

```
SUBROUTINE gauher(x,w,n)
INTEGER n,MAXIT
REAL w(n),x(n)
DOUBLE PRECISION EPS,PIM4
PARAMETER (EPS=3.D-14,PIM4=.7511255444649425D0,MAXIT=10)
    Given n, this routine returns arrays x(1:n) and w(1:n) containing the abscissas and
    weights of the n-point Gauss-Hermite quadrature formula. The largest abscissa is returned
    in x(1), the most negative in x(n).
    Parameters: EPS is the relative precision, PIM4 = 1/π^1/4, MAXIT = maximum iterations.
INTEGER i,its,j,m
DOUBLE PRECISION p1,p2,p3,pp,z,z1
  High precision is a good idea for this routine.
m=(n+1)/2
  The roots are symmetric about the origin, so we have to find only half of them.
do 13 i=1,m                          Loop over the desired roots.
    if(i.eq.1)then                   Initial guess for the largest root.
        z=sqrt(float(2*n+1))-1.85575*(2*n+1)**(-.16667)
    else if(i.eq.2)then              Initial guess for the second largest root.
        z=z-1.14*n**.426/z
    else if (i.eq.3)then             Initial guess for the third largest root.
        z=1.86*z-.86*x(1)
    else if (i.eq.4)then             Initial guess for the fourth largest root.
        z=1.91*z-.91*x(2)
    else                             Initial guess for the other roots.
        z=2.*z-x(i-2)
```

```
        endif
        do 12 its=1,MAXIT                Refinement by Newton's method.
            p1=PIM4
            p2=0.d0
            do 11 j=1,n                  Loop up the recurrence relation to get the Hermite poly-
                p3=p2                        nomial evaluated at z.
                p2=p1
                p1=z*sqrt(2.d0/j)*p2-sqrt(dble(j-1)/dble(j))*p3
            enddo 11
              p1 is now the desired Hermite polynomial. We next compute pp, its derivative, by
              the relation (4.5.21) using p2, the polynomial of one lower order.
            pp=sqrt(2.d0*n)*p2
            z1=z
            z=z1-p1/pp                   Newton's formula.
            if(abs(z-z1).le.EPS)goto 1
        enddo 12
        pause 'too many iterations in gauher'
1       x(i)=z                           Store the root
        x(n+1-i)=-z                      and its symmetric counterpart.
        w(i)=2.d0/(pp*pp)                Compute the weight
        w(n+1-i)=w(i)                    and its symmetric counterpart.
    enddo 13
    return
    END
```

Finally, here is a routine for Gauss-Jacobi abscissas and weights, which implement the integration formula

$$\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}f(x)dx=\sum_{j=1}^{N}w_j f(x_j) \tag{4.5.23}$$

```
    SUBROUTINE gaujac(x,w,n,alf,bet)
    INTEGER n,MAXIT
    REAL alf,bet,w(n),x(n)
    DOUBLE PRECISION EPS
    PARAMETER (EPS=3.D-14,MAXIT=10)      Increase EPS if you don't have this precision.
C   USES gammln
        Given alf and bet, the parameters α and β of the Jacobi polynomials, this routine returns
        arrays x(1:n) and w(1:n) containing the abscissas and weights of the n-point Gauss-Jacobi
        quadrature formula. The largest abscissa is returned in x(1), the smallest in x(n).
    INTEGER i,its,j
    REAL alfbet,an,bn,r1,r2,r3,gammln
    DOUBLE PRECISION a,b,c,p1,p2,p3,pp,temp,z,z1
      High precision is a good idea for this routine.
    do 13 i=1,n                          Loop over the desired roots.
        if(i.eq.1)then                   Initial guess for the largest root.
            an=alf/n
            bn=bet/n
            r1=(1.+alf)*(2.78/(4.+n*n)+.768*an/n)
            r2=1.+1.48*an+.96*bn+.452*an*an+.83*an*bn
            z=1.-r1/r2
        else if(i.eq.2)then              Initial guess for the second largest root.
            r1=(4.1+alf)/((1.+alf)*(1.+.156*alf))
            r2=1.+.06*(n-8.)*(1.+.12*alf)/n
            r3=1.+.012*bet*(1.+.25*abs(alf))/n
            z=z-(1.-z)*r1*r2*r3
        else if(i.eq.3)then              Initial guess for the third largest root.
            r1=(1.67+.28*alf)/(1.+.37*alf)
            r2=1.+.22*(n-8.)/n
```

```
            r3=1.+8.*bet/((6.28+bet)*n*n)
            z=z-(x(1)-z)*r1*r2*r3
        else if(i.eq.n-1)then           Initial guess for the second smallest root.
            r1=(1.+.235*bet)/(.766+.119*bet)
            r2=1./(1.+.639*(n-4.)/(1.+.71*(n-4.)))
            r3=1./(1.+20.*alf/((7.5+alf)*n*n))
            z=z+(z-x(n-3))*r1*r2*r3
        else if(i.eq.n)then             Initial guess for the smallest root.
            r1=(1.+.37*bet)/(1.67+.28*bet)
            r2=1./(1.+.22*(n-8.)/n)
            r3=1./(1.+8.*alf/((6.28+alf)*n*n))
            z=z+(z-x(n-2))*r1*r2*r3
        else                            Initial guess for the other roots.
            z=3.*x(i-1)-3.*x(i-2)+x(i-3)
        endif
        alfbet=alf+bet
        do 12 its=1,MAXIT               Refinement by Newton's method.
            temp=2.d0+alfbet            Start the recurrence with P_0 and P_1 to avoid a divi-
            p1=(alf-bet+temp*z)/2.d0        sion by zero when α + β = 0 or −1.
            p2=1.d0
            do 11 j=2,n                 Loop up the recurrence relation to get the Jacobi
                p3=p2                       polynomial evaluated at z.
                p2=p1
                temp=2*j+alfbet
                a=2*j*(j+alfbet)*(temp-2.d0)
                b=(temp-1.d0)*(alf*alf-bet*bet+temp*
     *              (temp-2.d0)*z)
                c=2.d0*(j-1+alf)*(j-1+bet)*temp
                p1=(b*p2-c*p3)/a
            enddo 11
            pp=(n*(alf-bet-temp*z)*p1+2.d0*(n+alf)*
     *           (n+bet)*p2)/(temp*(1.d0-z*z))
              p1 is now the desired Jacobi polynomial. We next compute pp, its derivative, by a
              standard relation involving also p2, the polynomial of one lower order.
            z1=z
            z=z1-p1/pp                  Newton's formula.
            if(abs(z-z1).le.EPS)goto 1
        enddo 12
        pause 'too many iterations in gaujac'
1       x(i)=z                          Store the root and the weight.
        w(i)=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.)-
     *       gammln(n+alfbet+1.))*temp*2.**alfbet/(pp*p2)
    enddo 13
    return
    END
```

Legendre polynomials are special cases of Jacobi polynomials with $\alpha = \beta = 0$, but it is worth having the separate routine for them, `gauleg`, given above. Chebyshev polynomials correspond to $\alpha = \beta = -1/2$ (see §5.8). They have analytic abscissas and weights:

$$x_j = \cos\left(\frac{\pi(j-\frac{1}{2})}{N}\right)$$
$$w_j = \frac{\pi}{N} \tag{4.5.24}$$

Case of Known Recurrences

Turn now to the case where you do not know good initial guesses for the zeros of your orthogonal polynomials, but you do have available the coefficients a_j and b_j that generate them. As we have seen, the zeros of $p_N(x)$ are the abscissas for the N-point Gaussian quadrature formula. The most useful computational formula for the weights is equation (4.5.9) above, since the derivative p'_N can be efficiently computed by the derivative of (4.5.6) in the general case, or by special relations for the classical polynomials. Note that (4.5.9) is valid as written only for monic polynomials; for other normalizations, there is an extra factor of λ_N/λ_{N-1}, where λ_N is the coefficient of x^N in p_N.

Except in those special cases already discussed, the best way to find the abscissas is *not* to use a root-finding method like Newton's method on $p_N(x)$. Rather, it is generally faster to use the Golub-Welsch [3] algorithm, which is based on a result of Wilf [4]. This algorithm notes that if you bring the term xp_j to the left-hand side of (4.5.6) and the term p_{j+1} to the right-hand side, the recurrence relation can be written in matrix form as

$$x\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 & & & & \\ b_1 & a_1 & 1 & & & \\ & \vdots & \vdots & & & \\ & & & b_{N-2} & a_{N-2} & 1 \\ & & & & b_{N-1} & a_{N-1} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ p_N \end{bmatrix}$$

or

$$x\mathbf{p} = \mathbf{T}\cdot\mathbf{p} + p_N\mathbf{e}_{N-1} \tag{4.5.25}$$

Here $\mathbf{T}$ is a tridiagonal matrix, $\mathbf{p}$ is a column vector of $p_0, p_1, \ldots, p_{N-1}$, and $\mathbf{e}_{N-1}$ is a unit vector with a 1 in the $(N-1)$st (last) position and zeros elsewhere. The matrix $\mathbf{T}$ can be symmetrized by a diagonal similarity transformation $\mathbf{D}$ to give

$$\mathbf{J} = \mathbf{DTD}^{-1} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \vdots & \vdots & & \\ & & \sqrt{b_{N-2}} & a_{N-2} & \sqrt{b_{N-1}} \\ & & & \sqrt{b_{N-1}} & a_{N-1} \end{bmatrix} \tag{4.5.26}$$

The matrix $\mathbf{J}$ is called the *Jacobi matrix* (not to be confused with other matrices named after Jacobi that arise in completely different problems!). Now we see from (4.5.25) that $p_N(x_j) = 0$ is equivalent to x_j being an eigenvalue of $\mathbf{T}$. Since eigenvalues are preserved by a similarity transformation, x_j is an eigenvalue of the symmetric tridiagonal matrix $\mathbf{J}$. Moreover, Wilf [4] shows that if $\mathbf{v}_j$ is the eigenvector corresponding to the eigenvalue x_j, normalized so that $\mathbf{v}\cdot\mathbf{v} = 1$, then

$$w_j = \mu_0 v_{j,1}^2 \tag{4.5.27}$$

where

$$\mu_0 = \int_a^b W(x)\,dx \tag{4.5.28}$$

and where $v_{j,1}$ is the first component of $\mathbf{v}$. As we shall see in Chapter 11, finding all eigenvalues and eigenvectors of a symmetric tridiagonal matrix is a relatively efficient and well-conditioned procedure. We accordingly give a routine, `gaucof`, for finding the abscissas and weights, given the coefficients a_j and b_j. Remember that if you know the recurrence relation for orthogonal polynomials that are not normalized to be monic, you can easily convert it to monic form by means of the quantities λ_j.

```
      SUBROUTINE gaucof(n,a,b,amu0,x,w)
      INTEGER n,NMAX
      REAL amu0,a(n),b(n),w(n),x(n)
      PARAMETER (NMAX=64)
C     USES eigsrt,tqli
```

Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi matrix. On input, `a(1:n)` and `b(1:n)` are the coefficients of the recurrence relation for the set of monic orthogonal polynomials. The quantity $\mu_0 \equiv \int_a^b W(x)\,dx$ is input as `amu0`. The abscissas `x(1:n)` are returned in descending order, with the corresponding weights in `w(1:n)`. The arrays `a` and `b` are modified. Execution can be speeded up by modifying `tqli` and `eigsrt` to compute only the first component of each eigenvector.

```
      INTEGER i,j
      REAL z(NMAX,NMAX)
      do 12 i=1,n
        if(i.ne.1)b(i)=sqrt(b(i))      Set up superdiagonal of Jacobi matrix.
        do 11 j=1,n                    Set up identity matrix for tqli to compute eigenvectors.
          if(i.eq.j)then
            z(i,j)=1.
          else
            z(i,j)=0.
          endif
        enddo 11
      enddo 12
      call tqli(a,b,n,NMAX,z)
      call eigsrt(a,z,n,NMAX)          Sort eigenvalues into descending order.
      do 13 i=1,n
        x(i)=a(i)
        w(i)=amu0*z(1,i)**2            Equation (4.5.12).
      enddo 13
      return
      END
```

Orthogonal Polynomials with Nonclassical Weights

This somewhat specialized subsection will tell you what to do if your weight function is not one of the classical ones dealt with above and you do not know the a_j's and b_j's of the recurrence relation (4.5.6) to use in `gaucof`. Then, a method of finding the a_j's and b_j's is needed.

The *procedure of Stieltjes* is to compute a_0 from (4.5.7), then $p_1(x)$ from (4.5.6). Knowing p_0 and p_1, we can compute a_1 and b_1 from (4.5.7), and so on. But how are we to compute the inner products in (4.5.7)?

The textbook approach is to represent each $p_j(x)$ explicitly as a polynomial in x and to compute the inner products by multiplying out term by term. This will be feasible if we know the first $2N$ moments of the weight function,

$$\mu_j = \int_a^b x^j W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.29}$$

However, the solution of the resulting set of algebraic equations for the coefficients a_j and b_j in terms of the moments μ_j is in general *extremely* ill-conditioned. Even in double precision, it is not unusual to lose all accuracy by the time $N = 12$. We thus reject any procedure based on the moments (4.5.29).

Sack and Donovan [5] discovered that the numerical stability is greatly improved if, instead of using powers of x as a set of basis functions to represent the p_j's, one uses some other known set of orthogonal polynomials $\pi_j(x)$, say. Roughly speaking, the improved stability occurs because the polynomial basis "samples" the interval (a, b) better than the power basis when the inner product integrals are evaluated, especially if its weight function resembles $W(x)$.

So assume that we know the *modified moments*

$$\nu_j = \int_a^b \pi_j(x) W(x) dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.30}$$

where the π_j's satisfy a recurrence relation analogous to (4.5.6),

$$\begin{aligned} \pi_{-1}(x) &\equiv 0 \\ \pi_0(x) &\equiv 1 \\ \pi_{j+1}(x) &= (x - \alpha_j)\pi_j(x) - \beta_j \pi_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.5.31}$$

and the coefficients α_j, β_j are known explicitly. Then Wheeler [6] has given an efficient $O(N^2)$ algorithm equivalent to that of Sack and Donovan for finding a_j and b_j via a set of intermediate quantities

$$\sigma_{k,l} = \langle p_k | \pi_l \rangle \qquad k, l \geq -1 \tag{4.5.32}$$

Initialize

$$\begin{aligned} \sigma_{-1,l} &= 0 && l = 1, 2, \ldots, 2N-2 \\ \sigma_{0,l} &= \nu_l && l = 0, 1, \ldots, 2N-1 \\ a_0 &= \alpha_0 + \frac{\nu_1}{\nu_0} \\ b_0 &= 0 \end{aligned} \tag{4.5.33}$$

Then, for $k = 1, 2, \ldots, N-1$, compute

$$\begin{aligned} \sigma_{k,l} &= \sigma_{k-1,l+1} - (a_{k-1} - \alpha_l)\sigma_{k-1,l} - b_{k-1}\sigma_{k-2,l} + \beta_l \sigma_{k-1,l-1} \\ &\qquad\qquad l = k, k+1, \ldots, 2N-k-1 \\ a_k &= \alpha_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}} \\ b_k &= \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}} \end{aligned} \tag{4.5.34}$$

Note that the normalization factors can also easily be computed if needed:

$$\begin{aligned} \langle p_0 | p_0 \rangle &= \nu_0 \\ \langle p_j | p_j \rangle &= b_j \langle p_{j-1} | p_{j-1} \rangle \qquad j = 1, 2, \ldots \end{aligned} \tag{4.5.35}$$

You can find a derivation of the above algorithm in Ref. [7].

Wheeler's algorithm requires that the modified moments (4.5.30) be accurately computed. In practical cases there is often a closed form, or else recurrence relations can be used. The algorithm is extremely successful for *finite* intervals (a, b). For infinite intervals, the algorithm does not completely remove the ill-conditioning. In this case, Gautschi [8,9] recommends reducing the interval to a finite interval by a change of variable, and then using a suitable discretization procedure to compute the inner products. You will have to consult the references for details.

We give the routine `orthog` for generating the coefficients a_j and b_j by Wheeler's algorithm, given the coefficients α_j and β_j, and the modified moments ν_j. To conform to the usual `FORTRAN` convention for dimensioning subscripts, the indices of the σ matrix are increased by 2, i.e., `sig(k,l)` $= \sigma_{k-2,l-2}$, while the indices of the vectors α, β, a and b are increased by 1.

```
      SUBROUTINE orthog(n,anu,alpha,beta,a,b)
      INTEGER n,NMAX
      REAL a(n),alpha(2*n-1),anu(2*n),b(n),beta(2*n-1)
      PARAMETER (NMAX=64)
        Computes the coefficients a_j and b_j, j = 0,...N-1, of the recurrence relation for
        monic orthogonal polynomials with weight function W(x) by Wheeler's algorithm. On input,
        alpha(1:2*n-1) and beta(1:2*n-1) are the coefficients α_j and β_j, j = 0,...2N-2,
        of the recurrence relation for the chosen basis of orthogonal polynomials. The modified
        moments ν_j are input in anu(1:2*n). The first n coefficients are returned in a(1:n) and
        b(1:n).
      INTEGER k,l
      REAL sig(2*NMAX+1,2*NMAX+1)
      do 11 l=3,2*n                    Initialization, Equation (4.5.33).
        sig(1,l)=0.
      enddo 11
      do 12 l=2,2*n+1
        sig(2,l)=anu(l-1)
      enddo 12
      a(1)=alpha(1)+anu(2)/anu(1)
      b(1)=0.
      do 14 k=3,n+1                    Equation (4.5.34).
        do 13 l=k,2*n-k+3
          sig(k,l)=sig(k-1,l+1)+(alpha(l-1)-a(k-2))*sig(k-1,l)-
     *        b(k-2)*sig(k-2,l)+beta(l-1)*sig(k-1,l-1)
        enddo 13
        a(k-1)=alpha(k-1)+sig(k,k+1)/sig(k,k)-sig(k-1,k)/sig(k-1,k-1)
        b(k-1)=sig(k,k)/sig(k-1,k-1)
      enddo 14
      return
      END
```

As an example of the use of `orthog`, consider the problem [7] of generating orthogonal polynomials with the weight function $W(x) = -\log x$ on the interval $(0, 1)$. A suitable set of π_j's is the shifted Legendre polynomials

$$\pi_j = \frac{(j!)^2}{(2j)!} P_j(2x-1) \tag{4.5.36}$$

The factor in front of P_j makes the polynomials monic. The coefficients in the recurrence relation (4.5.31) are

$$\begin{aligned} \alpha_j &= \frac{1}{2} & j &= 0, 1, \ldots \\ \beta_j &= \frac{1}{4(4-j^{-2})} & j &= 1, 2, \ldots \end{aligned} \tag{4.5.37}$$

while the modified moments are

$$\nu_j = \begin{cases} 1 & j = 0 \\ \dfrac{(-1)^j (j!)^2}{j(j+1)(2j)!} & j \geq 1 \end{cases} \tag{4.5.38}$$

A call to `orthog` with this input allows one to generate the required polynomials to machine accuracy for very large N, and hence do Gaussian quadrature with this weight function. Before Sack and Donovan's observation, this seemingly simple problem was essentially intractable.

Extensions of Gaussian Quadrature

There are many different ways in which the ideas of Gaussian quadrature have been extended. One important extension is the case of *preassigned nodes*: Some points are required to be included in the set of abscissas, and the problem is to choose

the weights and the remaining abscissas to maximize the degree of exactness of the the quadrature rule. The most common cases are *Gauss-Radau* quadrature, where one of the nodes is an endpoint of the interval, either a or b, and *Gauss-Lobatto* quadrature, where both a and b are nodes. Golub [10] has given an algorithm similar to `gaucof` for these cases.

The second important extension is the *Gauss-Kronrod* formulas. For ordinary Gaussian quadrature formulas, as N increases the sets of abscissas have no points in common. This means that if you compare results with increasing N as a way of estimating the quadrature error, you cannot reuse the previous function evaluations. Kronrod [11] posed the problem of searching for optimal sequences of rules, each of which reuses all abscissas of its predecessor. If one starts with $N = m$, say, and then adds n new points, one has $2n + m$ free parameters: the n new abscissas and weights, and m new weights for the fixed previous abscissas. The maximum degree of exactness one would expect to achieve would therefore be $2n + m - 1$. The question is whether this maximum degree of exactness can actually be achieved in practice, when the abscissas are required to all lie inside (a, b). The answer to this question is not known in general.

Kronrod showed that if you choose $n = m + 1$, an optimal extension can be found for Gauss-Legendre quadrature. Patterson [12] showed how to compute continued extensions of this kind. Sequences such as $N = 10, 21, 43, 87, \ldots$ are popular in automatic quadrature routines [13] that attempt to integrate a function until some specified accuracy has been achieved.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Golub, G.H., and Welsch, J.H. 1969, *Mathematics of Computation*, vol. 23, pp. 221–230 and A1–A10. [3]

Wilf, H.S. 1962, *Mathematics for the Physical Sciences* (New York: Wiley), Problem 9, p. 80. [4]

Sack, R.A., and Donovan, A.F. 1971/72, *Numerische Mathematik*, vol. 18, pp. 465–478. [5]

Wheeler, J.C. 1974, *Rocky Mountain Journal of Mathematics*, vol. 4, pp. 287–296. [6]

Gautschi, W. 1978, in *Recent Advances in Numerical Analysis*, C. de Boor and G.H. Golub, eds. (New York: Academic Press), pp. 45–72. [7]

Gautschi, W. 1981, in *E.B. Christoffel*, P.L. Butzer and F. Fehér, eds. (Basel: Birkhauser Verlag), pp. 72–147. [8]

Gautschi, W. 1990, in *Orthogonal Polynomials*, P. Nevai, ed. (Dordrecht: Kluwer Academic Publishers), pp. 181–216. [9]

Golub, G.H. 1973, *SIAM Review*, vol. 15, pp. 318–334. [10]

Kronrod, A.S. 1964, *Doklady Akademii Nauk SSSR*, vol. 154, pp. 283–286 (in Russian). [11]

Patterson, T.N.L. 1968, *Mathematics of Computation*, vol. 22, pp. 847–856 and C1–C11; 1969, *op. cit.*, vol. 23, p. 892. [12]

Piessens, R., de Doncker, E., Uberhuber, C.W., and Kahaner, D.K. 1983, *QUADPACK: A Subroutine Package for Automatic Integration* (New York: Springer-Verlag). [13]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.6.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §§2.9–2.10.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §§4.4–4.8.

4.6 Multidimensional Integrals

Integrals of functions of several variables, over regions with dimension greater than one, are *not easy*. There are two reasons for this. First, the number of function evaluations needed to sample an N-dimensional space increases as the Nth power of the number needed to do a one-dimensional integral. If you need 30 function evaluations to do a one-dimensional integral crudely, then you will likely need on the order of 30000 evaluations to reach the same crude level for a three-dimensional integral. Second, the region of integration in N-dimensional space is defined by an $N-1$ dimensional boundary which can itself be terribly complicated: It need not be convex or simply connected, for example. By contrast, the boundary of a one-dimensional integral consists of two numbers, its upper and lower limits.

The first question to be asked, when faced with a multidimensional integral, is, "can it be reduced analytically to a lower dimensionality?" For example, so-called *iterated integrals* of a function of one variable $f(t)$ can be reduced to one-dimensional integrals by the formula

$$\begin{aligned}\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1)dt_1 \\ = \frac{1}{(n-1)!}\int_0^x (x-t)^{n-1} f(t)dt\end{aligned} \tag{4.6.1}$$

Alternatively, the function may have some special symmetry in the way it depends on its independent variables. If the boundary also has this symmetry, then the dimension can be reduced. In three dimensions, for example, the integration of a spherically symmetric function over a spherical region reduces, in polar coordinates, to a one-dimensional integral.

The next questions to be asked will guide your choice between two entirely different approaches to doing the problem. The questions are: Is the shape of the boundary of the region of integration simple or complicated? Inside the region, is the integrand smooth and simple, or complicated, or locally strongly peaked? Does the problem require high accuracy, or does it require an answer accurate only to a percent, or a few percent?

If your answers are that the boundary is complicated, the integrand is *not* strongly peaked in very small regions, and relatively low accuracy is tolerable, then your problem is a good candidate for *Monte Carlo integration*. This method is very straightforward to program, in its cruder forms. One needs only to know a region with simple boundaries that *includes* the complicated region of integration, plus a method of determining whether a random point is inside or outside the region of integration. Monte Carlo integration evaluates the function at a random sample of

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §§2.9–2.10.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §§4.4–4.8.

4.6 Multidimensional Integrals

Integrals of functions of several variables, over regions with dimension greater than one, are *not easy*. There are two reasons for this. First, the number of function evaluations needed to sample an N-dimensional space increases as the Nth power of the number needed to do a one-dimensional integral. If you need 30 function evaluations to do a one-dimensional integral crudely, then you will likely need on the order of 30000 evaluations to reach the same crude level for a three-dimensional integral. Second, the region of integration in N-dimensional space is defined by an $N-1$ dimensional boundary which can itself be terribly complicated: It need not be convex or simply connected, for example. By contrast, the boundary of a one-dimensional integral consists of two numbers, its upper and lower limits.

The first question to be asked, when faced with a multidimensional integral, is, "can it be reduced analytically to a lower dimensionality?" For example, so-called *iterated integrals* of a function of one variable $f(t)$ can be reduced to one-dimensional integrals by the formula

$$\begin{aligned}\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1)dt_1 \\ = \frac{1}{(n-1)!}\int_0^x (x-t)^{n-1} f(t)dt\end{aligned} \tag{4.6.1}$$

Alternatively, the function may have some special symmetry in the way it depends on its independent variables. If the boundary also has this symmetry, then the dimension can be reduced. In three dimensions, for example, the integration of a spherically symmetric function over a spherical region reduces, in polar coordinates, to a one-dimensional integral.

The next questions to be asked will guide your choice between two entirely different approaches to doing the problem. The questions are: Is the shape of the boundary of the region of integration simple or complicated? Inside the region, is the integrand smooth and simple, or complicated, or locally strongly peaked? Does the problem require high accuracy, or does it require an answer accurate only to a percent, or a few percent?

If your answers are that the boundary is complicated, the integrand is *not* strongly peaked in very small regions, and relatively low accuracy is tolerable, then your problem is a good candidate for *Monte Carlo integration*. This method is very straightforward to program, in its cruder forms. One needs only to know a region with simple boundaries that *includes* the complicated region of integration, plus a method of determining whether a random point is inside or outside the region of integration. Monte Carlo integration evaluates the function at a random sample of

points, and estimates its integral based on that random sample. We will discuss it in more detail, and with more sophistication, in Chapter 7.

If the boundary is simple, and the function is very smooth, then the remaining approaches, breaking up the problem into repeated one-dimensional integrals, or multidimensional Gaussian quadratures, will be effective and relatively fast [1]. If you require high accuracy, these approaches are in any case the *only* ones available to you, since Monte Carlo methods are by nature asymptotically slow to converge.

For low accuracy, use repeated one-dimensional integration or multidimensional Gaussian quadratures when the integrand is slowly varying and smooth in the region of integration, Monte Carlo when the integrand is oscillatory or discontinuous, but not strongly peaked in small regions.

If the integrand *is* strongly peaked in small regions, and you know where those regions are, break the integral up into several regions so that the integrand is smooth in each, and do each separately. If you don't know where the strongly peaked regions are, you might as well (at the level of sophistication of this book) quit: It is hopeless to expect an integration routine to search out unknown pockets of large contribution in a huge N-dimensional space. (But see §7.8.)

If, on the basis of the above guidelines, you decide to pursue the repeated one-dimensional integration approach, here is how it works. For definiteness, we will consider the case of a three-dimensional integral in x, y, z-space. Two dimensions, or more than three dimensions, are entirely analogous.

The first step is to specify the region of integration by (i) its lower and upper limits in x, which we will denote x_1 and x_2; (ii) its lower and upper limits in y at a specified value of x, denoted $y_1(x)$ and $y_2(x)$; and (iii) its lower and upper limits in z at specified x and y, denoted $z_1(x, y)$ and $z_2(x, y)$. In other words, find the numbers x_1 and x_2, and the functions $y_1(x), y_2(x), z_1(x, y)$, and $z_2(x, y)$ such that

$$\begin{aligned} I &\equiv \int\int\int dx\, dy\, dz f(x, y, z) \\ &= \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dz\ f(x, y, z) \end{aligned} \tag{4.6.2}$$

For example, a two-dimensional integral over a circle of radius one centered on the origin becomes

$$\int_{-1}^{1} dx \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} dy\ f(x, y) \tag{4.6.3}$$

Now we can define a function $G(x, y)$ that does the innermost integral,

$$G(x, y) \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x, y, z)dz \tag{4.6.4}$$

and a function $H(x)$ that does the integral of $G(x, y)$,

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x, y)dy \tag{4.6.5}$$

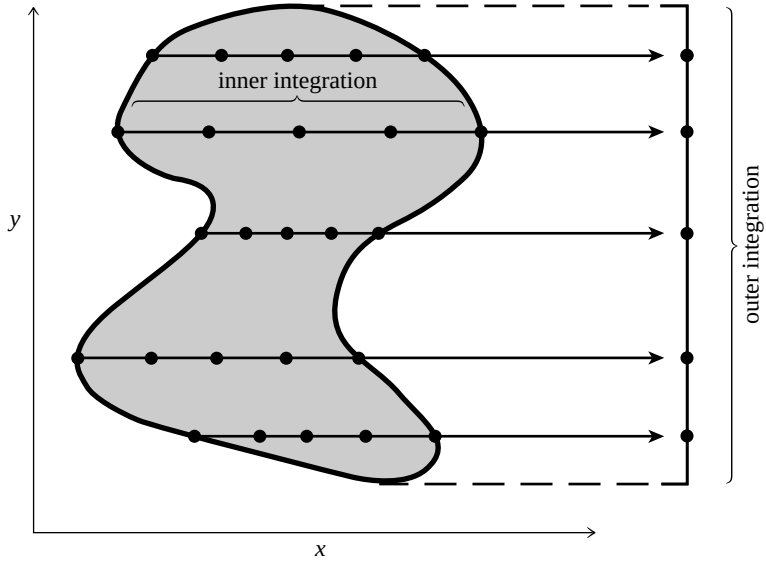


Figure 4.6.1. Function evaluations for a two-dimensional integral over an irregular region, shown schematically. The outer integration routine, in y, requests values of the inner, x, integral at locations along the y axis of its own choosing. The inner integration routine then evaluates the function at x locations suitable to *it*. This is more accurate in general than, e.g., evaluating the function on a Cartesian mesh of points.

and finally our answer as an integral over $H(x)$

$$I = \int_{x_1}^{x_2} H(x)dx \tag{4.6.6}$$

To implement equations (4.6.4)–(4.6.6) in a program, one needs three separate copies of a basic one-dimensional integration routine (and of any subroutines called by it), one each for the x, y, and z integrations. If you try to make do with only one copy, then it will call itself recursively, since (e.g.) the function evaluations of H for the x integration will themselves call the integration routine to do the y integration (see Figure 4.6.1). In our example, let us suppose that we plan to use the one-dimensional integrator qgaus of §4.5. Then we make three identical copies and call them qgausx, qgausy, and qgausz. The basic program for three-dimensional integration then is as follows:

```
      SUBROUTINE quad3d(x1,x2,ss)
      REAL ss,x1,x2,h
      EXTERNAL h
C     USES h,qgausx
        Returns as ss the integral of a user-supplied function func over a three-dimensional region
        specified by the limits x1, x2, and by the user-supplied functions y1, y2, z1, and z2, as
        defined in (4.6.2).
      call qgausx(h,x1,x2,ss)
      return
      END

      FUNCTION f(zz)
      REAL f,zz,func,x,y,z
```

```
      COMMON /xyz/ x,y,z
C     USES func
```
Called by qgausz. Calls func.
```
      z=zz
      f=func(x,y,z)
      return
      END

      FUNCTION g(yy)
      REAL g,yy,f,z1,z2,x,y,z
      EXTERNAL f
      COMMON /xyz/ x,y,z
C     USES f,qgausz,z1,z2
```
Called by qgausy. Calls qgausz.
```
      REAL ss
      y=yy
      call qgausz(f,z1(x,y),z2(x,y),ss)
      g=ss
      return
      END

      FUNCTION h(xx)
      REAL h,xx,g,y1,y2,x,y,z
      EXTERNAL g
      COMMON /xyz/ x,y,z
C     USES g,qgausy,y1,y2
```
Called by qgausx. Calls qgausy.
```
      REAL ss
      x=xx
      call qgausy(g,y1(x),y2(x),ss)
      h=ss
      return
      END
```

The necessary user-supplied functions have the following calling sequences:

```
FUNCTION func(x,y,z)        The 3-dimensional function to be integrated
FUNCTION y1(x)
FUNCTION y2(x)
FUNCTION z1(x,y)
FUNCTION z2(x,y)
```

CITED REFERENCES AND FURTHER READING:

Stroud, A.H. 1971, *Approximate Calculation of Multiple Integrals* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.7, p. 318.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.2.5, p. 307.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), equations 25.4.58ff.

Chapter 5. Evaluation of Functions

5.0 Introduction

The purpose of this chapter is to acquaint you with a selection of the techniques that are frequently used in evaluating functions. In Chapter 6, we will apply and illustrate these techniques by giving routines for a variety of specific functions. The purposes of this chapter and the next are thus mostly in harmony, but there is nevertheless some tension between them: Routines that are clearest and most illustrative of the general techniques of this chapter are not always the methods of choice for a particular special function. By comparing this chapter to the next one, you should get some idea of the balance between "general" and "special" methods that occurs in practice.

Insofar as that balance favors general methods, this chapter should give you ideas about how to write your own routine for the evaluation of a function which, while "special" to you, is not so special as to be included in Chapter 6 or the standard program libraries.

CITED REFERENCES AND FURTHER READING:

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall).

Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover), Chapter 7.

5.1 Series and Their Convergence

Everybody knows that an analytic function can be expanded in the neighborhood of a point x_0 in a power series,

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k \tag{5.1.1}$$

Such series are straightforward to evaluate. You don't, of course, evaluate the kth power of $x - x_0$ *ab initio* for each term; rather you keep the $k-1$st power and update it with a multiply. Similarly, the form of the coefficients a is often such as to make use of previous work: Terms like $k!$ or $(2k)!$ can be updated in a multiply or two.

Chapter 5. Evaluation of Functions

5.0 Introduction

The purpose of this chapter is to acquaint you with a selection of the techniques that are frequently used in evaluating functions. In Chapter 6, we will apply and illustrate these techniques by giving routines for a variety of specific functions. The purposes of this chapter and the next are thus mostly in harmony, but there is nevertheless some tension between them: Routines that are clearest and most illustrative of the general techniques of this chapter are not always the methods of choice for a particular special function. By comparing this chapter to the next one, you should get some idea of the balance between "general" and "special" methods that occurs in practice.

Insofar as that balance favors general methods, this chapter should give you ideas about how to write your own routine for the evaluation of a function which, while "special" to you, is not so special as to be included in Chapter 6 or the standard program libraries.

CITED REFERENCES AND FURTHER READING:

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall).

Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover), Chapter 7.

5.1 Series and Their Convergence

Everybody knows that an analytic function can be expanded in the neighborhood of a point x_0 in a power series,

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k \tag{5.1.1}$$

Such series are straightforward to evaluate. You don't, of course, evaluate the kth power of $x - x_0$ *ab initio* for each term; rather you keep the $k-1$st power and update it with a multiply. Similarly, the form of the coefficients a is often such as to make use of previous work: Terms like $k!$ or $(2k)!$ can be updated in a multiply or two.

How do you know when you have summed enough terms? In practice, the terms had better be getting small fast, otherwise the series is not a good technique to use in the first place. While not mathematically rigorous in all cases, standard practice is to quit when the term you have just added is smaller in magnitude than some small ϵ times the magnitude of the sum thus far accumulated. (But watch out if isolated instances of $a_k = 0$ are possible!).

A weakness of a power series representation is that it is guaranteed *not* to converge farther than that distance from x_0 at which a singularity is encountered *in the complex plane*. This catastrophe is not usually unexpected: When you find a power series in a book (or when you work one out yourself), you will generally also know the radius of convergence. An insidious problem occurs with series that converge everywhere (in the mathematical sense), but almost nowhere fast enough to be useful in a numerical method. Two familiar examples are the sine function and the Bessel function of the first kind,

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \tag{5.1.2}$$

$$J_n(x) = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!(k+n)!} \tag{5.1.3}$$

Both of these series converge for all x. But both don't even start to converge until $k \gg |x|$; before this, their terms are increasing. This makes these series useless for large x.

Accelerating the Convergence of Series

There are several tricks for accelerating the rate of convergence of a series (or, equivalently, of a sequence of partial sums). These tricks will *not* generally help in cases like (5.1.2) or (5.1.3) while the size of the terms is still increasing. For series with terms of decreasing magnitude, however, some accelerating methods can be startlingly good. *Aitken's δ^2-process* is simply a formula for extrapolating the partial sums of a series whose convergence is approximately geometric. If S_{n-1}, S_n, S_{n+1} are three successive partial sums, then an improved estimate is

$$S'_n \equiv S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n+1} - 2S_n + S_{n-1}} \tag{5.1.4}$$

You can also use (5.1.4) with $n+1$ and $n-1$ replaced by $n+p$ and $n-p$ respectively, for any integer p. If you form the sequence of S'_i's, you can apply (5.1.4) a second time to *that* sequence, and so on. (In practice, this iteration will only rarely do much for you after the first stage.) Note that equation (5.1.4) should be computed as written; there exist algebraically equivalent forms that are much more susceptible to roundoff error.

For *alternating series* (where the terms in the sum alternate in sign), *Euler's transformation* can be a powerful tool. Generally it is advisable to do a small

number $n-1$ of terms directly, then apply the transformation to the rest of the series beginning with the nth term. The formula (for n even) is

$$\sum_{s=0}^{\infty}(-1)^s u_s = u_0 - u_1 + u_2 \ldots - u_{n-1} + \sum_{s=0}^{\infty}\frac{(-1)^s}{2^{s+1}}[\Delta^s u_n] \tag{5.1.5}$$

Here Δ is the *forward difference operator*, i.e.,

$$\begin{aligned}\Delta u_n &\equiv u_{n+1} - u_n \\ \Delta^2 u_n &\equiv u_{n+2} - 2u_{n+1} + u_n \\ \Delta^3 u_n &\equiv u_{n+3} - 3u_{n+2} + 3u_{n+1} - u_n \qquad \text{etc.}\end{aligned} \tag{5.1.6}$$

Of course you don't actually do the infinite sum on the right-hand side of (5.1.5), but only the first, say, p terms, thus requiring the first p differences (5.1.6) obtained from the terms starting at u_n.

Euler's transformation can be applied not only to convergent series. In some cases it will produce accurate answers from the first terms of a series that is formally divergent. It is widely used in the summation of asymptotic series. In this case it is generally wise not to sum farther than where the terms start increasing in magnitude; and you should devise some independent numerical check that the results are meaningful.

There is an elegant and subtle implementation of Euler's transformation due to van Wijngaarden [1]: It incorporates the terms of the original alternating series one at a time, in order. For each incorporation it *either* increases p by 1, equivalent to computing one further difference (5.1.6), or else *retroactively* increases n by 1, without having to redo all the difference calculations based on the old n value! The decision as to which to increase, n or p, is taken in such a way as to make the convergence most rapid. Van Wijngaarden's technique requires only one vector of saved partial differences. Here is the algorithm:

```
SUBROUTINE eulsum(sum,term,jterm,wksp)
INTEGER jterm
REAL sum,term,wksp(jterm)          Workspace, provided by the calling program.
```

Incorporates into `sum` the `jterm`'th term, with value `term`, of an alternating series. `sum` is input as the previous partial sum, and is output as the new partial sum. The first call to this routine, with the first `term` in the series, should be with `jterm=1`. On the second call, `term` should be set to the second term of the series, with sign opposite to that of the first call, and `jterm` should be 2. And so on.

```
INTEGER j,nterm
REAL dum,tmp
SAVE nterm
if(jterm.eq.1)then                 Initialize:
    nterm=1                        Number of saved differences in wksp.
    wksp(1)=term
    sum=0.5*term                   Return first estimate.
else
    tmp=wksp(1)
    wksp(1)=term
    do 11 j=1,nterm-1              Update saved quantities by van Wijngaarden's algo-
        dum=wksp(j+1)                  rithm.
        wksp(j+1)=0.5*(wksp(j)+tmp)
        tmp=dum
```

```
    enddo 11
    wksp(nterm+1)=0.5*(wksp(nterm)+tmp)
    if(abs(wksp(nterm+1)).le.abs(wksp(nterm)))then      Favorable to increase p,
        sum=sum+0.5*wksp(nterm+1)
        nterm=nterm+1                  and the table becomes longer.
    else                               Favorable to increase n,
        sum=sum+wksp(nterm+1)          the table doesn't become longer.
    endif
endif
return
END
```

The powerful Euler technique is not directly applicable to a series of positive terms. Occasionally it is useful to convert a series of positive terms into an alternating series, just so that the Euler transformation can be used! Van Wijngaarden has given a transformation for accomplishing this [1]:

$$\sum_{r=1}^{\infty} v_r = \sum_{r=1}^{\infty} (-1)^{r-1} w_r \tag{5.1.7}$$

where

$$w_r \equiv v_r + 2v_{2r} + 4v_{4r} + 8v_{8r} + \cdots \tag{5.1.8}$$

Equations (5.1.7) and (5.1.8) replace a simple sum by a two-dimensional sum, each term in (5.1.7) being itself an infinite sum (5.1.8). This may seem a strange way to save on work! Since, however, the indices in (5.1.8) increase tremendously rapidly, as powers of 2, it often requires only a few terms to converge (5.1.8) to extraordinary accuracy. You do, however, need to be able to compute the v_r's efficiently for "random" values r. The standard "updating" tricks for sequential r's, mentioned above following equation (5.1.1), can't be used.

Actually, Euler's transformation is a special case of a more general transformation of power series. Suppose that some known function $g(z)$ has the series

$$g(z) = \sum_{n=0}^{\infty} b_n z^n \tag{5.1.9}$$

and that you want to sum the new, unknown, series

$$f(z) = \sum_{n=0}^{\infty} c_n b_n z^n \tag{5.1.10}$$

Then it is not hard to show (see [2]) that equation (5.1.10) can be written as

$$f(z) = \sum_{n=0}^{\infty} [\Delta^{(n)} c_0] \frac{g^{(n)}}{n!} z^n \tag{5.1.11}$$

which often converges much more rapidly. Here $\Delta^{(n)} c_0$ is the nth finite-difference operator (equation 5.1.6), with $\Delta^{(0)} c_0 \equiv c_0$, and $g^{(n)}$ is the nth derivative of $g(z)$. The usual Euler transformation (equation 5.1.5 with $n = 0$) can be obtained, for example, by substituting

$$g(z) = \frac{1}{1+z} = 1 - z + z^2 - z^3 + \cdots \tag{5.1.12}$$

into equation (5.1.11), and then setting $z = 1$.

Sometimes you will want to compute a function from a series representation even when the computation is *not* efficient. For example, you may be using the values obtained to fit the function to an approximating form that you will use subsequently (cf. §5.8). If you are summing very large numbers of slowly convergent terms, pay attention to roundoff errors! In floating-point representation it is more accurate to sum a list of numbers in the order starting with the smallest one, rather than starting with the largest one. It is even better to group terms pairwise, then in pairs of pairs, etc., so that all additions involve operands of comparable magnitude.

CITED REFERENCES AND FURTHER READING:

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 13 [van Wijngaarden's transformations]. [1]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 3.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §3.6.

Mathews, J., and Walker, R.L. 1970,

into equation (5.1.11), and then setting $z = 1$.

Sometimes you will want to compute a function from a series representation even when the computation is *not* efficient. For example, you may be using the values obtained to fit the function to an approximating form that you will use subsequently (cf. §5.8). If you are summing very large numbers of slowly convergent terms, pay attention to roundoff errors! In floating-point representation it is more accurate to sum a list of numbers in the order starting with the smallest one, rather than starting with the largest one. It is even better to group terms pairwise, then in pairs of pairs, etc., so that all additions involve operands of comparable magnitude.

CITED REFERENCES AND FURTHER READING:

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 13 [van Wijngaarden's transformations]. [1]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 3.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §3.6.

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), §2.3. [2]

5.2 Evaluation of Continued Fractions

Continued fractions are often powerful ways of evaluating functions that occur in scientific applications. A continued fraction looks like this:

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \cdots}}}}} \tag{5.2.1}$$

Printers prefer to write this as

$$f(x) = b_0 + \frac{a_1}{b_1 +} \; \frac{a_2}{b_2 +} \; \frac{a_3}{b_3 +} \; \frac{a_4}{b_4 +} \; \frac{a_5}{b_5 +} \cdots \tag{5.2.2}$$

In either (5.2.1) or (5.2.2), the a's and b's can themselves be functions of x, usually linear or quadratic monomials at worst (i.e., constants times x or times x^2). For example, the continued fraction representation of the tangent function is

$$\tan x = \frac{x}{1 -} \; \frac{x^2}{3 -} \; \frac{x^2}{5 -} \; \frac{x^2}{7 -} \cdots \tag{5.2.3}$$

Continued fractions frequently converge much more rapidly than power series expansions, and in a much larger domain in the complex plane (not necessarily including the domain of convergence of the series, however). Sometimes the continued fraction converges best where the series does worst, although this is not

a general rule. Blanch [1] gives a good review of the most useful convergence tests for continued fractions.

There are standard techniques, including the important *quotient-difference algorithm*, for going back and forth between continued fraction approximations, power series approximations, and rational function approximations. Consult Acton [2] for an introduction to this subject, and Fike [3] for further details and references.

How do you tell how far to go when evaluating a continued fraction? Unlike a series, you can't just evaluate equation (5.2.1) from left to right, stopping when the change is small. Written in the form of (5.2.1), the only way to evaluate the continued fraction is from right to left, first (blindly!) guessing how far out to start. This is not the right way.

The right way is to use a result that relates continued fractions to rational approximations, and that gives a means of evaluating (5.2.1) or (5.2.2) from left to right. Let f_n denote the result of evaluating (5.2.2) with coefficients through a_n and b_n. Then

$$f_n = \frac{A_n}{B_n} \tag{5.2.4}$$

where A_n and B_n are given by the following recurrence:

$$\begin{aligned} A_{-1} &\equiv 1 \qquad B_{-1} \equiv 0 \\ A_0 &\equiv b_0 \qquad B_0 \equiv 1 \\ A_j = b_j A_{j-1} + a_j A_{j-2} &\qquad B_j = b_j B_{j-1} + a_j B_{j-2} \qquad j = 1, 2, \ldots, n \end{aligned} \tag{5.2.5}$$

This method was invented by J. Wallis in 1655 (!), and is discussed in his *Arithmetica Infinitorum* [4]. You can easily prove it by induction.

In practice, this algorithm has some unattractive features: The recurrence (5.2.5) frequently generates very large or very small values for the partial numerators and denominators A_j and B_j. There is thus the danger of overflow or underflow of the floating-point representation. However, the recurrence (5.2.5) is linear in the A's and B's. At any point you can rescale the currently saved two levels of the recurrence, e.g., divide A_j, B_j, A_{j-1}, and B_{j-1} all by B_j. This incidentally makes $A_j = f_j$ and is convenient for testing whether you have gone far enough: See if f_j and f_{j-1} from the last iteration are as close as you would like them to be. (If B_j happens to be zero, which can happen, just skip the renormalization for this cycle. A fancier level of optimization is to renormalize only when an overflow is imminent, saving the unnecessary divides. All this complicates the program logic.)

Two newer algorithms have been proposed for evaluating continued fractions. *Steed's method* does not use A_j and B_j explicitly, but only the *ratio* $D_j = B_{j-1}/B_j$. One calculates D_j and $\Delta f_j = f_j - f_{j-1}$ recursively using

$$D_j = 1/(b_j + a_j D_{j-1}) \tag{5.2.6}$$

$$\Delta f_j = (b_j D_j - 1)\Delta f_{j-1} \tag{5.2.7}$$

Steed's method (see, e.g., [5]) avoids the need for rescaling of intermediate results. However, for certain continued fractions you can occasionally run into a situation

where the denominator in (5.2.6) approaches zero, so that D_j and Δf_j are very large. The next Δf_{j+1} will typically cancel this large change, but with loss of accuracy in the numerical running sum of the f_j's. It is awkward to program around this, so Steed's method can be recommended only for cases where you know in advance that no denominator can vanish. We will use it for a special purpose in the routine `bessik` (§6.7).

The best general method for evaluating continued fractions seems to be the *modified Lentz's method* [6]. The need for rescaling intermediate results is avoided by using *both* the ratios

$$C_j = A_j/A_{j-1}, \qquad D_j = B_{j-1}/B_j \tag{5.2.8}$$

and calculating f_j by

$$f_j = f_{j-1}C_jD_j \tag{5.2.9}$$

From equation (5.2.5), one easily shows that the ratios satisfy the recurrence relations

$$D_j = 1/(b_j + a_jD_{j-1}), \qquad C_j = b_j + a_j/C_{j-1} \tag{5.2.10}$$

In this algorithm there is the danger that the denominator in the expression for D_j, or the quantity C_j itself, might approach zero. Either of these conditions invalidates (5.2.10). However, Thompson and Barnett [5] show how to modify Lentz's algorithm to fix this: Just shift the offending term by a small amount, e.g., 10^{-30}. If you work through a cycle of the algorithm with this prescription, you will see that f_{j+1} is accurately calculated.

In detail, the modified Lentz's algorithm is this:

- Set $f_0 = b_0$; if $b_0 = 0$ set $f_0 = tiny$.
- Set $C_0 = f_0$.
- Set $D_0 = 0$.
- For $j = 1, 2, \ldots$
  - Set $D_j = b_j + a_jD_{j-1}$.
  - If $D_j = 0$, set $D_j = tiny$.
  - Set $C_j = b_j + a_j/C_{j-1}$.
  - If $C_j = 0$ set $C_j = tiny$.
  - Set $D_j = 1/D_j$.
  - Set $\Delta_j = C_jD_j$.
  - Set $f_j = f_{j-1}\Delta_j$.
  - If $|\Delta_j - 1| < eps$ then exit.

Here eps is your floating-point precision, say 10^{-7} or 10^{-15}. The parameter $tiny$ should be less than typical values of $eps|b_j|$, say 10^{-30}.

The above algorithm assumes that you can terminate the evaluation of the continued fraction when $|f_j - f_{j-1}|$ is sufficiently small. This is usually the case, but by no means guaranteed. Jones [7] gives a list of theorems that can be used to justify this termination criterion for various kinds of continued fractions.

There is at present no rigorous analysis of error propagation in Lentz's algorithm. However, empirical tests suggest that it is at least as good as other methods.

Manipulating Continued Fractions

Several important properties of continued fractions can be used to rewrite them in forms that can speed up numerical computation. An *equivalence transformation*

$$a_n \to \lambda a_n, \quad b_n \to \lambda b_n, \quad a_{n+1} \to \lambda a_{n+1} \tag{5.2.11}$$

leaves the value of a continued fraction unchanged. By a suitable choice of the scale factor λ you can often simplify the form of the a's and the b's. Of course, you can carry out successive equivalence transformations, possibly with different λ's, on successive terms of the continued fraction.

The *even* and *odd* parts of a continued fraction are continued fractions whose successive convergents are f_{2n} and f_{2n+1}, respectively. Their main use is that they converge twice as fast as the original continued fraction, and so if their terms are not much more complicated than the terms in the original there can be a big savings in computation. The formula for the even part of (5.2.2) is

$$f_{\rm even} = d_0 + \frac{c_1}{d_1 +} \; \frac{c_2}{d_2 +} \; \cdots \tag{5.2.12}$$

where in terms of intermediate variables

$$\begin{aligned} \alpha_1 &= \frac{a_1}{b_1} \\ \alpha_n &= \frac{a_n}{b_n b_{n-1}}, \qquad n \ge 2 \end{aligned} \tag{5.2.13}$$

we have

$$\begin{aligned} d_0 = b_0, \quad c_1 = \alpha_1, \quad d_1 = 1 + \alpha_2 \\ c_n = -\alpha_{2n-1}\alpha_{2n-2}, \quad d_n = 1 + \alpha_{2n-1} + \alpha_{2n}, \qquad n \ge 2 \end{aligned} \tag{5.2.14}$$

You can find the similar formula for the odd part in the review by Blanch [1]. Often a combination of the transformations (5.2.14) and (5.2.11) is used to get the best form for numerical work.

We will make frequent use of continued fractions in the next chapter.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §3.10.

Blanch, G. 1964, *SIAM Review*, vol. 6, pp. 383–421. [1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 11. [2]

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 1.

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), §§8.2, 10.4, and 10.5. [3]

Wallis, J. 1695, in *Opera Mathematica*, vol. 1, p. 355, Oxoniae e Theatro Shedoniano. Reprinted by Georg Olms Verlag, Hildeshein, New York (1972). [4]

Thompson, I.J., and Barnett, A.R. 1986, *Journal of Computational Physics*, vol. 64, pp. 490–509. [5]

Lentz, W.J. 1976, *Applied Optics*, vol. 15, pp. 668–671. [6]

Jones, W.B. 1973, in *Padé Approximants and Their Applications*, P.R. Graves-Morris, ed. (London: Academic Press), p. 125. [7]

5.3 Polynomials and Rational Functions

A polynomial of degree $N-1$ is represented numerically as a stored array of coefficients, `c(j)` with `j`$= 1, \ldots, N$. We will always take `c(1)` to be the constant term in the polynomial, `c(`N`)` the coefficient of x^{N-1}; but of course other conventions are possible. There are two kinds of manipulations that you can do with a polynomial: *numerical* manipulations (such as evaluation), where you are given the numerical value of its argument, or *algebraic* manipulations, where you want to transform the coefficient array in some way without choosing any particular argument. Let's start with the numerical.

We assume that you know enough *never* to evaluate a polynomial this way:

```
p=c(1)+c(2)*x+c(3)*x**2+c(4)*x**3+c(5)*x**4
```

Come the (computer) revolution, all persons found guilty of such criminal behavior will be summarily executed, and their programs won't be! It is a matter of taste, however, whether to write

```
p=c(1)+x*(c(2)+x*(c(3)+x*(c(4)+x*c(5))))
```

or

```
p=(((c(5)*x+c(4))*x+c(3))*x+c(2))*x+c(1)
```

If the number of coefficients is a large number `n`, one writes

```
p=c(n)
do 11 j=n-1,1,-1
    p=p*x+c(j)
enddo 11
```

Another useful trick is for evaluating a polynomial $P(x)$ and its derivative $dP(x)/dx$ simultaneously:

```
p=c(n)
dp=0.
do 11 j=n-1,1,-1
    dp=dp*x+p
    p=p*x+c(j)
enddo 11
```

which returns the polynomial as `p` and its derivative as `dp`.

The above trick, which is basically *synthetic division* [1,2], generalizes to the evaluation of the polynomial and `nd-1` of its derivatives simultaneously:

Thompson, I.J., and Barnett, A.R. 1986, *Journal of Computational Physics*, vol. 64, pp. 490–509. [5]

Lentz, W.J. 1976, *Applied Optics*, vol. 15, pp. 668–671. [6]

Jones, W.B. 1973, in *Padé Approximants and Their Applications*, P.R. Graves-Morris, ed. (London: Academic Press), p. 125. [7]

5.3 Polynomials and Rational Functions

A polynomial of degree $N-1$ is represented numerically as a stored array of coefficients, `c(j)` with `j`$= 1, \ldots, N$. We will always take `c(1)` to be the constant term in the polynomial, `c(`N`)` the coefficient of x^{N-1}; but of course other conventions are possible. There are two kinds of manipulations that you can do with a polynomial: *numerical* manipulations (such as evaluation), where you are given the numerical value of its argument, or *algebraic* manipulations, where you want to transform the coefficient array in some way without choosing any particular argument. Let's start with the numerical.

We assume that you know enough *never* to evaluate a polynomial this way:

```
p=c(1)+c(2)*x+c(3)*x**2+c(4)*x**3+c(5)*x**4
```

Come the (computer) revolution, all persons found guilty of such criminal behavior will be summarily executed, and their programs won't be! It is a matter of taste, however, whether to write

```
p=c(1)+x*(c(2)+x*(c(3)+x*(c(4)+x*c(5))))
```

or

```
p=(((c(5)*x+c(4))*x+c(3))*x+c(2))*x+c(1)
```

If the number of coefficients is a large number `n`, one writes

```
p=c(n)
do 11 j=n-1,1,-1
    p=p*x+c(j)
enddo 11
```

Another useful trick is for evaluating a polynomial $P(x)$ and its derivative $dP(x)/dx$ simultaneously:

```
p=c(n)
dp=0.
do 11 j=n-1,1,-1
    dp=dp*x+p
    p=p*x+c(j)
enddo 11
```

which returns the polynomial as `p` and its derivative as `dp`.

The above trick, which is basically *synthetic division* [1,2], generalizes to the evaluation of the polynomial and `nd-1` of its derivatives simultaneously:

```
SUBROUTINE ddpoly(c,nc,x,pd,nd)
INTEGER nc,nd
REAL x,c(nc),pd(nd)
    Given the coefficients of a polynomial of degree nc-1 as an array c(1:nc) with c(1) being
    the constant term, and given a value x, and given a value nd>1, this routine returns the
    polynomial evaluated at x as pd(1) and nd-1 derivatives as pd(2:nd).
INTEGER i,j,nnd
REAL const
pd(1)=c(nc)
do 11 j=2,nd
    pd(j)=0.
enddo 11
do 13 i=nc-1,1,-1
    nnd=min(nd,nc+1-i)
    do 12 j=nnd,2,-1
        pd(j)=pd(j)*x+pd(j-1)
    enddo 12
    pd(1)=pd(1)*x+c(i)
enddo 13
const=2.                          After the first derivative, factorial constants come in.
do 14 i=3,nd
    pd(i)=const*pd(i)
    const=const*i
enddo 14
return
END
```

As a curiosity, you might be interested to know that polynomials of degree $n > 3$ can be evaluated in *fewer* than n multiplications, at least if you are willing to precompute some auxiliary coefficients and, in some cases, do an extra addition. For example, the polynomial

$$P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 \tag{5.3.1}$$

where $a_4 > 0$, can be evaluated with 3 multiplications and 5 additions as follows:

$$P(x) = [(Ax + B)^2 + Ax + C][(Ax + B)^2 + D] + E \tag{5.3.2}$$

where A, B, C, D, and E are to be precomputed by

$$\begin{aligned} A &= (a_4)^{1/4} \\ B &= \frac{a_3 - A^3}{4A^3} \\ D &= 3B^2 + 8B^3 + \frac{a_1 A - 2a_2 B}{A^2} \\ C &= \frac{a_2}{A^2} - 2B - 6B^2 - D \\ E &= a_0 - B^4 - B^2(C + D) - CD \end{aligned} \tag{5.3.3}$$

Fifth degree polynomials can be evaluated in 4 multiplies and 5 adds; sixth degree polynomials can be evaluated in 4 multiplies and 7 adds; if any of this strikes you as interesting, consult references [3-5]. The subject has something of the same entertaining, if impractical, flavor as that of fast matrix multiplication, discussed in §2.11.

Turn now to algebraic manipulations. You multiply a polynomial of degree $n-1$ (array of length n) by a monomial factor $x-a$ by a bit of code like the following,

```
c(n+1)=c(n)
do 11 j=n,2,-1
    c(j)=c(j-1)-c(j)*a
enddo 11
c(1)=-c(1)*a
```

Likewise, you divide a polynomial of degree $n-1$ by a monomial factor $x-a$ (synthetic division again) using

```
rem=c(n)
c(n)=0.
do 11 i=n-1,1,-1
    swap=c(i)
    c(i)=rem
    rem=swap+rem*a
enddo 11
```

which leaves you with a new polynomial array and a numerical remainder `rem`.

Multiplication of two general polynomials involves straightforward summing of the products, each involving one coefficient from each polynomial. Division of two general polynomials, while it can be done awkwardly in the fashion taught using pencil and paper, is susceptible to a good deal of streamlining. Witness the following routine based on the algorithm in [3].

```
SUBROUTINE poldiv(u,n,v,nv,q,r)
INTEGER n,nv
REAL q(n),r(n),u(n),v(nv)
    Given the n coefficients of a polynomial in u(1:n), and the nv coefficients of another
    polynomial in v(1:nv), divide the polynomial u by the polynomial v ("u"/"v") giving
    a quotient polynomial whose coefficients are returned in q(1:n-nv+1), and a remainder
    polynomial whose coefficients are returned in r(1:nv-1). The arrays q and r are dimen-
    sioned with lengths n, but the elements r(nv)...r(n) and q(n-nv+2)...q(n) will be
    returned as zero.
INTEGER j,k
do 11 j=1,n
    r(j)=u(j)
    q(j)=0.
enddo 11
do 13 k=n-nv,0,-1
    q(k+1)=r(nv+k)/v(nv)
    do 12 j=nv+k-1,k+1,-1
        r(j)=r(j)-q(k+1)*v(j-k)
    enddo 12
enddo 13
do 14 j=nv,n
    r(j)=0.
enddo 14
return
END
```

Rational Functions

You evaluate a rational function like

$$R(x) = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \tag{5.3.4}$$

in the obvious way, namely as two separate polynomials followed by a divide. As a matter of convention one usually chooses $q_0 = 1$, obtained by dividing numerator and denominator by any other q_0. It is often convenient to have both sets of coefficients stored in a single array, and to have a standard subroutine available for doing the evaluation:

```
FUNCTION ratval(x,cof,mm,kk)
INTEGER kk,mm
DOUBLE PRECISION ratval,x,cof(mm+kk+1)     Note precision! Change to REAL if desired.
    Given mm, kk, and cof(1:mm+kk+1), evaluate and return the rational function (cof(1)+
    cof(2)x+···+cof(mm+1)x^mm)/(1+cof(mm+2)x+···+cof(mm+kk+1)x^kk).
INTEGER j
DOUBLE PRECISION sumd,sumn
sumn=cof(mm+1)
do 11 j=mm,1,-1
    sumn=sumn*x+cof(j)
enddo 11
sumd=0.d0
do 12 j=mm+kk+1,mm+2,-1
    sumd=(sumd+cof(j))*x
enddo 12
ratval=sumn/(1.d0+sumd)
return
END
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 183, 190. [1]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 361–363. [2]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6. [3]

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 4.

Winograd, S. 1970, *Communications on Pure and Applied Mathematics*, vol. 23, pp. 165–179. [4]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley). [5]

5.4 Complex Arithmetic

Since FORTRAN has the built-in data type COMPLEX, you can generally let the compiler and intrinsic function library take care of complex arithmetic for you. Generally, but not always. For a program with only a small number of complex operations, you may want to code these yourself, in-line. Or, you may find that your compiler is not up to snuff: It is disconcertingly common to encounter complex operations that produce overflows or underflows when both the complex operands and the complex result are perfectly representable. This occurs, we think, because software companies assign inexperienced programmers to what they believe to be the perfectly trivial task of implementing complex arithmetic.

Actually, complex arithmetic is not *quite* trivial. Addition and subtraction are done in the obvious way, performing the operation separately on the real and imaginary parts of the operands. Multiplication can also be done in the obvious way, with 4 multiplications, one addition, and one subtraction,

$$(a+ib)(c+id) = (ac-bd) + i(bc+ad) \tag{5.4.1}$$

(the addition before the i doesn't count; it just separates the real and imaginary parts notationally). But it is sometimes faster to multiply via

$$(a+ib)(c+id) = (ac-bd) + i[(a+b)(c+d) - ac - bd] \tag{5.4.2}$$

which has only three multiplications (ac, bd, $(a+b)(c+d)$), plus two additions and three subtractions. The total operations count is higher by two, but multiplication is a slow operation on some machines.

While it is true that intermediate results in equations (5.4.1) and (5.4.2) can overflow even when the final result is representable, this happens only when the final answer is on the edge of representability. Not so for the complex modulus, if you or your compiler are misguided enough to compute it as

$$|a+ib| = \sqrt{a^2+b^2} \qquad \text{(bad!)} \tag{5.4.3}$$

whose intermediate result will overflow if either a or b is as large as the square root of the largest representable number (e.g., 10^{19} as compared to 10^{38}). The right way to do the calculation is

$$|a+ib| = \begin{cases} |a|\sqrt{1+(b/a)^2} & |a| \geq |b| \\ |b|\sqrt{1+(a/b)^2} & |a| < |b| \end{cases} \tag{5.4.4}$$

Complex division should use a similar trick to prevent avoidable overflows, underflow, or loss of precision,

$$\frac{a+ib}{c+id} = \begin{cases} \dfrac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(d/c)} & |c| \geq |d| \\ \dfrac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c| < |d| \end{cases} \tag{5.4.5}$$

5.4 Complex Arithmetic

Since FORTRAN has the built-in data type COMPLEX, you can generally let the compiler and intrinsic function library take care of complex arithmetic for you. Generally, but not always. For a program with only a small number of complex operations, you may want to code these yourself, in-line. Or, you may find that your compiler is not up to snuff: It is disconcertingly common to encounter complex operations that produce overflows or underflows when both the complex operands and the complex result are perfectly representable. This occurs, we think, because software companies assign inexperienced programmers to what they believe to be the perfectly trivial task of implementing complex arithmetic.

Actually, complex arithmetic is not *quite* trivial. Addition and subtraction are done in the obvious way, performing the operation separately on the real and imaginary parts of the operands. Multiplication can also be done in the obvious way, with 4 multiplications, one addition, and one subtraction,

$$(a+ib)(c+id) = (ac-bd) + i(bc+ad) \tag{5.4.1}$$

(the addition before the i doesn't count; it just separates the real and imaginary parts notationally). But it is sometimes faster to multiply via

$$(a+ib)(c+id) = (ac-bd) + i[(a+b)(c+d) - ac - bd] \tag{5.4.2}$$

which has only three multiplications (ac, bd, $(a+b)(c+d)$), plus two additions and three subtractions. The total operations count is higher by two, but multiplication is a slow operation on some machines.

While it is true that intermediate results in equations (5.4.1) and (5.4.2) can overflow even when the final result is representable, this happens only when the final answer is on the edge of representability. Not so for the complex modulus, if you or your compiler are misguided enough to compute it as

$$|a+ib| = \sqrt{a^2+b^2} \qquad \text{(bad!)} \tag{5.4.3}$$

whose intermediate result will overflow if either a or b is as large as the square root of the largest representable number (e.g., 10^{19} as compared to 10^{38}). The right way to do the calculation is

$$|a+ib| = \begin{cases} |a|\sqrt{1+(b/a)^2} & |a| \geq |b| \\ |b|\sqrt{1+(a/b)^2} & |a| < |b| \end{cases} \tag{5.4.4}$$

Complex division should use a similar trick to prevent avoidable overflows, underflow, or loss of precision,

$$\frac{a+ib}{c+id} = \begin{cases} \dfrac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(d/c)} & |c| \geq |d| \\ \dfrac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c| < |d| \end{cases} \tag{5.4.5}$$

Of course you should calculate repeated subexpressions, like c/d or d/c, only once.

Complex square root is even more complicated, since we must both guard intermediate results, and also enforce a chosen branch cut (here taken to be the negative real axis). To take the square root of $c + id$, first compute

$$w \equiv \begin{cases} 0 & c = d = 0 \\ \sqrt{|c|}\sqrt{\dfrac{1+\sqrt{1+(d/c)^2}}{2}} & |c| \geq |d| \\ \sqrt{|d|}\sqrt{\dfrac{|c/d|+\sqrt{1+(c/d)^2}}{2}} & |c| < |d| \end{cases} \tag{5.4.6}$$

Then the answer is

$$\sqrt{c+id} = \begin{cases} 0 & w = 0 \\ w + i\left(\dfrac{d}{2w}\right) & w \neq 0, c \geq 0 \\ \dfrac{|d|}{2w} + iw & w \neq 0, c < 0, d \geq 0 \\ \dfrac{|d|}{2w} - iw & w \neq 0, c < 0, d < 0 \end{cases} \tag{5.4.7}$$

CITED REFERENCES AND FURTHER READING:

Midy, P., and Yakovlev, Y. 1991, *Mathematics and Computers in Simulation*, vol. 33, pp. 33–49.

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley) [see solutions to exercises 4.2.1.16 and 4.6.4.41].

5.5 Recurrence Relations and Clenshaw's Recurrence Formula

Many useful functions satisfy recurrence relations, e.g.,

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \tag{5.5.1}$$

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{5.5.2}$$

$$nE_{n+1}(x) = e^{-x} - xE_n(x) \tag{5.5.3}$$

$$\cos n\theta = 2\cos\theta\cos(n-1)\theta - \cos(n-2)\theta \tag{5.5.4}$$

$$\sin n\theta = 2\cos\theta\sin(n-1)\theta - \sin(n-2)\theta \tag{5.5.5}$$

where the first three functions are Legendre polynomials, Bessel functions of the first kind, and exponential integrals, respectively. (For notation see [1].) These relations

Of course you should calculate repeated subexpressions, like c/d or d/c, only once.

Complex square root is even more complicated, since we must both guard intermediate results, and also enforce a chosen branch cut (here taken to be the negative real axis). To take the square root of $c + id$, first compute

$$w \equiv \begin{cases} 0 & c = d = 0 \\ \sqrt{|c|}\sqrt{\dfrac{1+\sqrt{1+(d/c)^2}}{2}} & |c| \geq |d| \\ \sqrt{|d|}\sqrt{\dfrac{|c/d|+\sqrt{1+(c/d)^2}}{2}} & |c| < |d| \end{cases} \tag{5.4.6}$$

Then the answer is

$$\sqrt{c+id} = \begin{cases} 0 & w = 0 \\ w + i\left(\dfrac{d}{2w}\right) & w \neq 0, c \geq 0 \\ \dfrac{|d|}{2w} + iw & w \neq 0, c < 0, d \geq 0 \\ \dfrac{|d|}{2w} - iw & w \neq 0, c < 0, d < 0 \end{cases} \tag{5.4.7}$$

CITED REFERENCES AND FURTHER READING:

Midy, P., and Yakovlev, Y. 1991, *Mathematics and Computers in Simulation*, vol. 33, pp. 33–49.

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley) [see solutions to exercises 4.2.1.16 and 4.6.4.41].

5.5 Recurrence Relations and Clenshaw's Recurrence Formula

Many useful functions satisfy recurrence relations, e.g.,

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \tag{5.5.1}$$

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{5.5.2}$$

$$nE_{n+1}(x) = e^{-x} - xE_n(x) \tag{5.5.3}$$

$$\cos n\theta = 2\cos\theta\cos(n-1)\theta - \cos(n-2)\theta \tag{5.5.4}$$

$$\sin n\theta = 2\cos\theta\sin(n-1)\theta - \sin(n-2)\theta \tag{5.5.5}$$

where the first three functions are Legendre polynomials, Bessel functions of the first kind, and exponential integrals, respectively. (For notation see [1].) These relations

are useful for extending computational methods from two successive values of n to other values, either larger or smaller.

Equations (5.5.4) and (5.5.5) motivate us to say a few words about trigonometric functions. If your program's running time is dominated by evaluating trigonometric functions, you are probably doing something wrong. Trig functions whose arguments form a linear sequence $\theta = \theta_0 + n\delta$, $n = 0, 1, 2, \ldots$, are efficiently calculated by the following recurrence,

$$\begin{aligned} \cos(\theta + \delta) &= \cos\theta - [\alpha\cos\theta + \beta\sin\theta] \\ \sin(\theta + \delta) &= \sin\theta - [\alpha\sin\theta - \beta\cos\theta] \end{aligned} \tag{5.5.6}$$

where α and β are the precomputed coefficients

$$\alpha \equiv 2\sin^2\left(\frac{\delta}{2}\right) \qquad \beta \equiv \sin\delta \tag{5.5.7}$$

The reason for doing things this way, rather than with the standard (and equivalent) identities for sums of angles, is that here α and β do not lose significance if the incremental δ is small. Likewise, the adds in equation (5.5.6) should be done in the order indicated by square brackets. We will use (5.5.6) repeatedly in Chapter 12, when we deal with Fourier transforms.

Another trick, occasionally useful, is to note that both $\sin\theta$ and $\cos\theta$ can be calculated via a single call to $\tan$:

$$t \equiv \tan\left(\frac{\theta}{2}\right) \qquad \cos\theta = \frac{1 - t^2}{1 + t^2} \qquad \sin\theta = \frac{2t}{1 + t^2} \tag{5.5.8}$$

The cost of getting both $\sin$ and $\cos$, if you need them, is thus the cost of $\tan$ plus 2 multiplies, 2 divides, and 2 adds. On machines with slow trig functions, this can be a savings. *However*, note that special treatment is required if $\theta \to \pm\pi$. And also note that many modern machines have *very fast* trig functions; so you should not assume that equation (5.5.8) is faster without testing.

Stability of Recurrences

You need to be aware that recurrence relations are not necessarily *stable* against roundoff error in the direction that you propose to go (either increasing n or decreasing n). A three-term linear recurrence relation

$$y_{n+1} + a_n y_n + b_n y_{n-1} = 0, \qquad n = 1, 2, \ldots \tag{5.5.9}$$

has two linearly independent solutions, f_n and g_n say. Only one of these corresponds to the sequence of functions f_n that you are trying to generate. The other one g_n *may* be exponentially growing in the direction that you want to go, or exponentially damped, or exponentially neutral (growing or dying as some power law, for example). If it is exponentially growing, then the recurrence relation is of little or no practical use in that direction. This is the case, e.g., for (5.5.2) in the direction of increasing n, when $x < n$. You cannot generate Bessel functions of high n by forward recurrence on (5.5.2).

To state things a bit more formally, if

$$f_n/g_n \to 0 \quad \text{as} \quad n \to \infty \tag{5.5.10}$$

then f_n is called the *minimal* solution of the recurrence relation (5.5.9). Nonminimal solutions like g_n are called *dominant* solutions. The minimal solution is unique, if it exists, but dominant solutions are not — you can add an arbitrary multiple of f_n to a given g_n. You can evaluate any dominant solution by forward recurrence, *but not the minimal solution*. (Unfortunately it is sometimes the one you want.)

Abramowitz and Stegun (in their Introduction) [1] give a list of recurrences that are stable in the increasing or decreasing directions. That list does not contain all possible formulas, of course. Given a recurrence relation for some function $f_n(x)$ you can test it yourself with about five minutes of (human) labor: For a fixed x in your range of interest, start the recurrence not with true values of $f_j(x)$ and $f_{j+1}(x)$, but (first) with the values 1 and 0, respectively, and then (second) with 0 and 1, respectively. Generate 10 or 20 terms of the recursive sequences in the direction that you want to go (increasing or decreasing from j), for each of the two starting conditions. Look at the difference between the corresponding members of the two sequences. If the differences stay of order unity (absolute value less than 10, say), then the recurrence is stable. If they increase slowly, then the recurrence may be mildly unstable but quite tolerably so. If they increase catastrophically, then there is an exponentially growing solution of the recurrence. If you know that the function that you want actually corresponds to the growing solution, then you can keep the recurrence formula anyway e.g., the case of the Bessel function $Y_n(x)$ for increasing n, see §6.5; if you don't know which solution your function corresponds to, you must at this point reject the recurrence formula. Notice that you can do this test *before* you go to the trouble of finding a numerical method for computing the two starting functions $f_j(x)$ and $f_{j+1}(x)$: stability is a property of the recurrence, not of the starting values.

An alternative heuristic procedure for testing stability is to replace the recurrence relation by a similar one that is linear with constant coefficients. For example, the relation (5.5.2) becomes

$$y_{n+1} - 2\gamma y_n + y_{n-1} = 0 \tag{5.5.11}$$

where $\gamma \equiv n/x$ is treated as a constant. You solve such recurrence relations by trying solutions of the form $y_n = a^n$. Substituting into the above recurrence gives

$$a^2 - 2\gamma a + 1 = 0 \qquad \text{or} \qquad a = \gamma \pm \sqrt{\gamma^2 - 1} \tag{5.5.12}$$

The recurrence is stable if $|a| \le 1$ for all solutions a. This holds (as you can verify) if $|\gamma| \le 1$ or $n \le x$. The recurrence (5.5.2) thus cannot be used, starting with $J_0(x)$ and $J_1(x)$, to compute $J_n(x)$ for large n.

Possibly you would at this point like the security of some real theorems on this subject (although we ourselves always follow one of the heuristic procedures). Here are two theorems, due to Perron [2]:

Theorem A. If in (5.5.9) $a_n \sim an^\alpha$, $b_n \sim bn^\beta$ as $n \to \infty$, and $\beta < 2\alpha$, then

$$g_{n+1}/g_n \sim -an^\alpha, \qquad f_{n+1}/f_n \sim -(b/a)n^{\beta-\alpha} \tag{5.5.13}$$

and f_n is the minimal solution to (5.5.9).

Theorem B. Under the same conditions as Theorem A, but with $\beta = 2\alpha$, consider the *characteristic polynomial*

$$t^2 + at + b = 0 \tag{5.5.14}$$

If the roots t_1 and t_2 of (5.5.14) have distinct moduli, $|t_1| > |t_2|$ say, then

$$g_{n+1}/g_n \sim t_1 n^\alpha, \qquad f_{n+1}/f_n \sim t_2 n^\alpha \tag{5.5.15}$$

and f_n is again the minimal solution to (5.5.9). Cases other than those in these two theorems are inconclusive for the existence of minimal solutions. (For more on the stability of recurrences, see [3].)

How do you proceed if the solution that you desire *is* the minimal solution? The answer lies in that old aphorism, that every cloud has a silver lining: If a recurrence relation is catastrophically unstable in one direction, then that (undesired) solution will decrease very rapidly in the reverse direction. This means that you can start with *any* seed values for the consecutive f_j and f_{j+1} and (when you have gone enough steps in the stable direction) you will converge to the sequence of functions that you want, times an unknown normalization factor. If there is some other way to normalize the sequence (e.g., by a formula for the sum of the f_n's), then this can be a practical means of function evaluation. The method is called *Miller's algorithm*. An example often given [1,4] uses equation (5.5.2) in just this way, along with the normalization formula

$$1 = J_0(x) + 2J_2(x) + 2J_4(x) + 2J_6(x) + \cdots \tag{5.5.16}$$

Incidentally, there is an important relation between three-term recurrence relations and *continued fractions*. Rewrite the recurrence relation (5.5.9) as

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n + y_{n+1}/y_n} \tag{5.5.17}$$

Iterating this equation, starting with n, gives

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n -}\,\frac{b_{n+1}}{a_{n+1} -} \cdots \tag{5.5.18}$$

Pincherle's Theorem [2] tells us that (5.5.18) converges if and only if (5.5.9) has a minimal solution f_n, in which case it converges to f_n/f_{n-1}. This result, usually for the case $n = 1$ and combined with some way to determine f_0, underlies many of the practical methods for computing special functions that we give in the next chapter.

Clenshaw's Recurrence Formula

Clenshaw's recurrence formula [5] is an elegant and efficient way to evaluate a sum of coefficients times functions that obey a recurrence formula, e.g.,

$$f(\theta) = \sum_{k=0}^{N} c_k \cos k\theta \qquad \text{or} \qquad f(x) = \sum_{k=0}^{N} c_k P_k(x)$$

Here is how it works: Suppose that the desired sum is

$$f(x) = \sum_{k=0}^{N} c_k F_k(x) \tag{5.5.19}$$

and that F_k obeys the recurrence relation

$$F_{n+1}(x) = \alpha(n,x)F_n(x) + \beta(n,x)F_{n-1}(x) \tag{5.5.20}$$

for some functions $\alpha(n,x)$ and $\beta(n,x)$. Now define the quantities y_k $(k = N, N-1, \ldots, 1)$ by the following recurrence:

$$\begin{aligned} y_{N+2} &= y_{N+1} = 0 \\ y_k &= \alpha(k,x)y_{k+1} + \beta(k+1,x)y_{k+2} + c_k \quad (k = N, N-1, \ldots, 1) \end{aligned} \tag{5.5.21}$$

If you solve equation (5.5.21) for c_k on the left, and then write out explicitly the sum (5.5.19), it will look (in part) like this:

$$\begin{aligned} f(x) = &\cdots \\ &+ [y_8 - \alpha(8,x)y_9 - \beta(9,x)y_{10}]F_8(x) \\ &+ [y_7 - \alpha(7,x)y_8 - \beta(8,x)y_9]F_7(x) \\ &+ [y_6 - \alpha(6,x)y_7 - \beta(7,x)y_8]F_6(x) \\ &+ [y_5 - \alpha(5,x)y_6 - \beta(6,x)y_7]F_5(x) \\ &+ \cdots \\ &+ [y_2 - \alpha(2,x)y_3 - \beta(3,x)y_4]F_2(x) \\ &+ [y_1 - \alpha(1,x)y_2 - \beta(2,x)y_3]F_1(x) \\ &+ [c_0 + \beta(1,x)y_2 - \beta(1,x)y_2]F_0(x) \end{aligned} \tag{5.5.22}$$

Notice that we have added and subtracted $\beta(1,x)y_2$ in the last line. If you examine the terms containing a factor of y_8 in (5.5.22), you will find that they sum to zero as a consequence of the recurrence relation (5.5.20); similarly all the other y_k's down through y_2. The only surviving terms in (5.5.22) are

$$f(x) = \beta(1,x)F_0(x)y_2 + F_1(x)y_1 + F_0(x)c_0 \tag{5.5.23}$$

Equations (5.5.21) and (5.5.23) are *Clenshaw's recurrence formula* for doing the sum (5.5.19): You make one pass down through the y_k's using (5.5.21); when you have reached y_2 and y_1 you apply (5.5.23) to get the desired answer.

Clenshaw's recurrence as written above incorporates the coefficients c_k in a downward order, with k decreasing. At each stage, the effect of all previous c_k's is "remembered" as two coefficients which multiply the functions F_{k+1} and F_k (ultimately F_0 and F_1). If the functions F_k are small when k is large, *and* if the coefficients c_k are small when k is *small*, then the sum can be dominated by small F_k's. In this case the remembered coefficients will involve a delicate cancellation and there can be a catastrophic loss of significance. An example would be to sum the trivial series

$$J_{15}(1) = 0 \times J_0(1) + 0 \times J_1(1) + \ldots + 0 \times J_{14}(1) + 1 \times J_{15}(1) \tag{5.5.24}$$

Here J_{15}, which is tiny, ends up represented as a canceling linear combination of J_0 and J_1, which are of order unity.

The solution in such cases is to use an alternative Clenshaw recurrence that incorporates c_k's in an upward direction. The relevant equations are

$$y_{-2} = y_{-1} = 0 \tag{5.5.25}$$

$$y_k = \frac{1}{\beta(k+1,x)}[y_{k-2} - \alpha(k,x)y_{k-1} - c_k], \qquad (k = 0, 1, \ldots, N-1) \tag{5.5.26}$$

$$f(x) = c_N F_N(x) - \beta(N,x)F_{N-1}(x)y_{N-1} - F_N(x)y_{N-2} \tag{5.5.27}$$

The rare case where equations (5.5.25)–(5.5.27) should be used instead of equations (5.5.21) and (5.5.23) can be detected automatically by testing whether the operands in the first sum in (5.5.23) are opposite in sign and nearly equal in magnitude. Other than in this special case, Clenshaw's recurrence is always stable, independent of whether the recurrence for the functions F_k is stable in the upward or downward direction.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), pp. xiii, 697. [1]

Gautschi, W. 1967, *SIAM Review*, vol. 9, pp. 24–82. [2]

Lakshmikantham, V., and Trigiante, D. 1988, *Theory of Difference Equations: Numerical Methods and Applications* (San Diego: Academic Press). [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 20ff. [4]

Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory (London: H.M. Stationery Office). [5]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §4.4.3, p. 111.

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), p. 76.

5.6 Quadratic and Cubic Equations

The roots of simple algebraic equations can be viewed as being functions of the equations' coefficients. We are taught these functions in elementary algebra. Yet, surprisingly many people don't know the right way to solve a quadratic equation with two real roots, or to obtain the roots of a cubic equation.

There are two ways to write the solution of the *quadratic equation*

$$ax^2 + bx + c = 0 \tag{5.6.1}$$

with real coefficients a, b, c, namely

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.6.2}$$

and

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \tag{5.6.3}$$

If you use *either* (5.6.2) *or* (5.6.3) to get the two roots, you are asking for trouble: If either a or c (or both) are small, then one of the roots will involve the subtraction of b from a very nearly equal quantity (the discriminant); you will get that root very inaccurately. The correct way to compute the roots is

$$q \equiv -\frac{1}{2}\left[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right] \tag{5.6.4}$$

Then the two roots are

$$x_1 = \frac{q}{a} \qquad \text{and} \qquad x_2 = \frac{c}{q} \tag{5.6.5}$$

If the coefficients a, b, c, are complex rather than real, then the above formulas still hold, except that in equation (5.6.4) the sign of the square root should be chosen so as to make

$$\mathrm{Re}(b^* \sqrt{b^2 - 4ac}) \geq 0 \tag{5.6.6}$$

where Re denotes the real part and asterisk denotes complex conjugation.

Apropos of quadratic equations, this seems a convenient place to recall that the inverse hyperbolic functions $\sinh^{-1}$ and $\cosh^{-1}$ are in fact just logarithms of solutions to such equations,

$$\sinh^{-1}(x) = \quad \ln\left(x + \sqrt{x^2 + 1}\right) \tag{5.6.7}$$

$$\cosh^{-1}(x) = \pm \ln\left(x + \sqrt{x^2 - 1}\right) \tag{5.6.8}$$

Equation (5.6.7) is numerically robust for $x \geq 0$. For negative x, use the symmetry $\sinh^{-1}(-x) = -\sinh^{-1}(x)$. Equation (5.6.8) is of course valid only for $x \geq 1$. Since `FORTRAN` mysteriously omits the inverse hyperbolic functions from its list of intrinsic functions, equations (5.6.7)–(5.6.8) are sometimes quite essential.

5.6 Quadratic and Cubic Equations

The roots of simple algebraic equations can be viewed as being functions of the equations' coefficients. We are taught these functions in elementary algebra. Yet, surprisingly many people don't know the right way to solve a quadratic equation with two real roots, or to obtain the roots of a cubic equation.

There are two ways to write the solution of the *quadratic equation*

$$ax^2 + bx + c = 0 \tag{5.6.1}$$

with real coefficients a, b, c, namely

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.6.2}$$

and

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \tag{5.6.3}$$

If you use *either* (5.6.2) *or* (5.6.3) to get the two roots, you are asking for trouble: If either a or c (or both) are small, then one of the roots will involve the subtraction of b from a very nearly equal quantity (the discriminant); you will get that root very inaccurately. The correct way to compute the roots is

$$q \equiv -\frac{1}{2}\left[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right] \tag{5.6.4}$$

Then the two roots are

$$x_1 = \frac{q}{a} \qquad \text{and} \qquad x_2 = \frac{c}{q} \tag{5.6.5}$$

If the coefficients a, b, c, are complex rather than real, then the above formulas still hold, except that in equation (5.6.4) the sign of the square root should be chosen so as to make

$$\mathrm{Re}(b^* \sqrt{b^2 - 4ac}) \geq 0 \tag{5.6.6}$$

where Re denotes the real part and asterisk denotes complex conjugation.

Apropos of quadratic equations, this seems a convenient place to recall that the inverse hyperbolic functions $\sinh^{-1}$ and $\cosh^{-1}$ are in fact just logarithms of solutions to such equations,

$$\sinh^{-1}(x) = \quad \ln\left(x + \sqrt{x^2 + 1}\right) \tag{5.6.7}$$

$$\cosh^{-1}(x) = \pm \ln\left(x + \sqrt{x^2 - 1}\right) \tag{5.6.8}$$

Equation (5.6.7) is numerically robust for $x \geq 0$. For negative x, use the symmetry $\sinh^{-1}(-x) = -\sinh^{-1}(x)$. Equation (5.6.8) is of course valid only for $x \geq 1$. Since `FORTRAN` mysteriously omits the inverse hyperbolic functions from its list of intrinsic functions, equations (5.6.7)–(5.6.8) are sometimes quite essential.

For the *cubic equation*

$$x^3 + ax^2 + bx + c = 0 \tag{5.6.9}$$

with real or complex coefficients a, b, c, first compute

$$Q \equiv \frac{a^2 - 3b}{9} \qquad \text{and} \qquad R \equiv \frac{2a^3 - 9ab + 27c}{54} \tag{5.6.10}$$

If Q and R are real (always true when a, b, c are real) *and* $R^2 < Q^3$, then the cubic equation has three real roots. Find them by computing

$$\theta = \arccos(R/\sqrt{Q^3}) \tag{5.6.11}$$

in terms of which the three roots are

$$\begin{aligned} x_1 &= -2\sqrt{Q}\cos\left(\frac{\theta}{3}\right) - \frac{a}{3} \\ x_2 &= -2\sqrt{Q}\cos\left(\frac{\theta + 2\pi}{3}\right) - \frac{a}{3} \\ x_3 &= -2\sqrt{Q}\cos\left(\frac{\theta - 2\pi}{3}\right) - \frac{a}{3} \end{aligned} \tag{5.6.12}$$

(This equation first appears in Chapter VI of François Viète's treatise "De emendatione," published in 1615!)

Otherwise, compute

$$A = -\left[R + \sqrt{R^2 - Q^3}\right]^{1/3} \tag{5.6.13}$$

where the sign of the square root is chosen to make

$$\mathrm{Re}(R^*\sqrt{R^2 - Q^3}) \geq 0 \tag{5.6.14}$$

(asterisk again denoting complex conjugation). If Q and R are both real, equations (5.6.13)–(5.6.14) are equivalent to

$$A = -\mathrm{sgn}(R)\left[|R| + \sqrt{R^2 - Q^3}\right]^{1/3} \tag{5.6.15}$$

where the positive square root is assumed. Next compute

$$B = \begin{cases} Q/A & (A \neq 0) \\ 0 & (A = 0) \end{cases} \tag{5.6.16}$$

in terms of which the three roots are

$$x_1 = (A + B) - \frac{a}{3} \tag{5.6.17}$$

(the single real root when a, b, c are real) and

$$x_2 = -\frac{1}{2}(A+B) - \frac{a}{3} + i\frac{\sqrt{3}}{2}(A-B)$$
$$x_3 = -\frac{1}{2}(A+B) - \frac{a}{3} - i\frac{\sqrt{3}}{2}(A-B) \tag{5.6.18}$$

(in that same case, a complex conjugate pair). Equations (5.6.13)–(5.6.16) are arranged both to minimize roundoff error, and also (as pointed out by A.J. Glassman) to ensure that no choice of branch for the complex cube root can result in the spurious loss of a distinct root.

If you need to solve many cubic equations with only slightly different coefficients, it is more efficient to use Newton's method (§9.4).

CITED REFERENCES AND FURTHER READING:

Weast, R.C. (ed.) 1967, *Handbook of Tables for Mathematics*, 3rd ed. (Cleveland: The Chemical Rubber Co.), pp. 130–133.

Pachner, J. 1983, *Handbook of Numerical Analysis Applications* (New York: McGraw-Hill), §6.1.

McKelvey, J.P. 1984, *American Journal of Physics*, vol. 52, pp. 269–270; see also vol. 53, p. 775, and vol. 55, pp. 374–375.

5.7 Numerical Derivatives

Imagine that you have a procedure which computes a function $f(x)$, and now you want to compute its derivative $f'(x)$. Easy, right? The definition of the derivative, the limit as $h \to 0$ of

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{5.7.1}$$

practically suggests the program: Pick a small value h; evaluate $f(x+h)$; you probably have $f(x)$ already evaluated, but if not, do it too; finally apply equation (5.7.1). What more needs to be said?

Quite a lot, actually. Applied uncritically, the above procedure is almost guaranteed to produce inaccurate results. Applied properly, it can be the right way to compute a derivative only when the function f is *fiercely* expensive to compute, when you already have invested in computing $f(x)$, and when, therefore, you want to get the derivative in no more than a single additional function evaluation. In such a situation, the remaining issue is to choose h properly, an issue we now discuss:

There are two sources of error in equation (5.7.1), truncation error and roundoff error. The truncation error comes from higher terms in the Taylor series expansion,

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \cdots \tag{5.7.2}$$

whence

$$\frac{f(x+h) - f(x)}{h} = f' + \frac{1}{2}hf'' + \cdots \tag{5.7.3}$$

(the single real root when a, b, c are real) and

$$x_2 = -\frac{1}{2}(A+B) - \frac{a}{3} + i\frac{\sqrt{3}}{2}(A-B)$$
$$x_3 = -\frac{1}{2}(A+B) - \frac{a}{3} - i\frac{\sqrt{3}}{2}(A-B) \tag{5.6.18}$$

(in that same case, a complex conjugate pair). Equations (5.6.13)–(5.6.16) are arranged both to minimize roundoff error, and also (as pointed out by A.J. Glassman) to ensure that no choice of branch for the complex cube root can result in the spurious loss of a distinct root.

If you need to solve many cubic equations with only slightly different coefficients, it is more efficient to use Newton's method (§9.4).

CITED REFERENCES AND FURTHER READING:

Weast, R.C. (ed.) 1967, *Handbook of Tables for Mathematics*, 3rd ed. (Cleveland: The Chemical Rubber Co.), pp. 130–133.

Pachner, J. 1983, *Handbook of Numerical Analysis Applications* (New York: McGraw-Hill), §6.1.

McKelvey, J.P. 1984, *American Journal of Physics*, vol. 52, pp. 269–270; see also vol. 53, p. 775, and vol. 55, pp. 374–375.

5.7 Numerical Derivatives

Imagine that you have a procedure which computes a function $f(x)$, and now you want to compute its derivative $f'(x)$. Easy, right? The definition of the derivative, the limit as $h \to 0$ of

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{5.7.1}$$

practically suggests the program: Pick a small value h; evaluate $f(x+h)$; you probably have $f(x)$ already evaluated, but if not, do it too; finally apply equation (5.7.1). What more needs to be said?

Quite a lot, actually. Applied uncritically, the above procedure is almost guaranteed to produce inaccurate results. Applied properly, it can be the right way to compute a derivative only when the function f is *fiercely* expensive to compute, when you already have invested in computing $f(x)$, and when, therefore, you want to get the derivative in no more than a single additional function evaluation. In such a situation, the remaining issue is to choose h properly, an issue we now discuss:

There are two sources of error in equation (5.7.1), truncation error and roundoff error. The truncation error comes from higher terms in the Taylor series expansion,

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \cdots \tag{5.7.2}$$

whence

$$\frac{f(x+h) - f(x)}{h} = f' + \frac{1}{2}hf'' + \cdots \tag{5.7.3}$$

The roundoff error has various contributions. First there is roundoff error in h: Suppose, by way of an example, that you are at a point $x = 10.3$ and you blindly choose $h = 0.0001$. Neither $x = 10.3$ nor $x + h = 10.30001$ is a number with an exact representation in binary; each is therefore represented with some fractional error characteristic of the machine's floating-point format, ϵ_m, whose value in single precision may be $\sim 10^{-7}$. The error in the *effective* value of h, namely the difference between $x + h$ and x as represented in the machine, is therefore on the order of $\epsilon_m x$, which implies a fractional error in h of order $\sim \epsilon_m x/h \sim 10^{-2}$! By equation (5.7.1) this immediately implies at least the same large fractional error in the derivative.

We arrive at Lesson 1: Always choose h so that $x + h$ and x differ by an exactly representable number. This can usually be accomplished by the program steps

$$\begin{aligned} \texttt{temp} &= x + h \\ h &= \texttt{temp} - x \end{aligned} \tag{5.7.4}$$

Some optimizing compilers, and some computers whose floating-point chips have higher internal accuracy than is stored externally, can foil this trick; if so, it is usually enough to call a dummy subroutine `donothing(temp)` between the two equations (5.7.4). This forces `temp` into and out of addressable memory.

With h an "exact" number, the roundoff error in equation (5.7.1) is $e_r \sim \epsilon_f |f(x)/h|$. Here ϵ_f is the fractional accuracy with which f is computed; for a simple function this may be comparable to the machine accuracy, $\epsilon_f \approx \epsilon_m$, but for a complicated calculation with additional sources of inaccuracy it may be larger. The truncation error in equation (5.7.3) is on the order of $e_t \sim |hf''(x)|$. Varying h to minimize the sum $e_r + e_t$ gives the optimal choice of h,

$$h \sim \sqrt{\frac{\epsilon_f f}{f''}} \approx \sqrt{\epsilon_f}\, x_c \tag{5.7.5}$$

where $x_c \equiv (f/f'')^{1/2}$ is the "curvature scale" of the function f, or "characteristic scale" over which it changes. In the absence of any other information, one often assumes $x_c = x$ (except near $x = 0$ where some other estimate of the typical x scale should be used).

With the choice of equation (5.7.5), the fractional accuracy of the computed derivative is

$$(e_r + e_t)/|f'| \sim \sqrt{\epsilon_f}(ff''/f'^2)^{1/2} \sim \sqrt{\epsilon_f} \tag{5.7.6}$$

Here the last order-of-magnitude equality assumes that f, f', and f'' all share the same characteristic length scale, usually the case. One sees that the simple finite-difference equation (5.7.1) gives *at best* only the square root of the machine accuracy ϵ_m.

If you can afford two function evaluations for each derivative calculation, then it is significantly better to use the symmetrized form

$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h} \tag{5.7.7}$$

In this case, by equation (5.7.2), the truncation error is $e_t \sim h^2 f'''$. The roundoff error e_r is about the same as before. The optimal choice of h, by a short calculation analogous to the one above, is now

$$h \sim \left(\frac{\epsilon_f f}{f'''} \right)^{1/3} \sim (\epsilon_f)^{1/3} x_c \tag{5.7.8}$$

and the fractional error is

$$(e_r + e_t)/|f'| \sim (\epsilon_f)^{2/3} f^{2/3} (f''')^{1/3}/f' \sim (\epsilon_f)^{2/3} \tag{5.7.9}$$

which will typically be an order of magnitude (single precision) or two orders of magnitude (double precision) *better* than equation (5.7.6). We have arrived at Lesson 2: Choose h to be *the correct* power of ϵ_f or ϵ_m times a characteristic scale x_c.

You can easily derive the correct powers for other cases [1]. For a function of two dimensions, for example, and the mixed derivative formula

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{[f(x+h, y+h) - f(x+h, y-h)] - [f(x-h, y+h) - f(x-h, y-h)]}{4h^2} \tag{5.7.10}$$

the correct scaling is $h \sim \epsilon_f^{1/3} x_c$.

It is disappointing, certainly, that no simple finite-difference formula like equation (5.7.1) or (5.7.7) gives an accuracy comparable to the machine accuracy ϵ_m, or even the lower accuracy to which f is evaluated, ϵ_f. Are there no better methods?

Yes, there are. All, however, involve exploration of the function's behavior over scales comparable to x_c, plus some assumption of smoothness, or analyticity, so that the high-order terms in a Taylor expansion like equation (5.7.2) have some meaning. Such methods also involve multiple evaluations of the function f, so their increased accuracy must be weighed against increased cost.

The general idea of "Richardson's deferred approach to the limit" is particularly attractive. For numerical integrals, that idea leads to so-called Romberg integration (for review, see §4.3). For derivatives, one seeks to extrapolate, to $h \to 0$, the result of finite-difference calculations with smaller and smaller finite values of h. By the use of Neville's algorithm (§3.1), one uses each new finite-difference calculation to produce both an extrapolation of higher order, and also extrapolations of previous, lower, orders but with smaller scales h. Ridders [2] has given a nice implementation of this idea; the following program, `dfridr`, is based on his algorithm, modified by an improved termination criterion. Input to the routine is a function f (called `func`), a position x, and a *largest* stepsize `h` (more analogous to what we have called x_c above than to what we have called h). Output is the returned value of the derivative, and an estimate of its error, `err`.

```
      FUNCTION dfridr(func,x,h,err)
      INTEGER NTAB
      REAL dfridr,err,h,x,func,CON,CON2,BIG,SAFE
      PARAMETER (CON=1.4,CON2=CON*CON,BIG=1.E30,NTAB=10,SAFE=2.)
      EXTERNAL func
C     USES func
        Returns the derivative of a function func at a point x by Ridders' method of polynomial
        extrapolation. The value h is input as an estimated initial stepsize; it need not be small,
```

```
    but rather should be an increment in x over which func changes substantially. An estimate
    of the error in the derivative is returned as err.
    Parameters: Stepsize is decreased by CON at each iteration. Max size of tableau is set by
    NTAB. Return when error is SAFE worse than the best so far.
INTEGER i,j
REAL errt,fac,hh,a(NTAB,NTAB)
if(h.eq.0.) pause 'h must be nonzero in dfridr'
hh=h
a(1,1)=(func(x+hh)-func(x-hh))/(2.0*hh)
err=BIG
do 12 i=2,NTAB                  Successive columns in the Neville tableau will go to smaller
    hh=hh/CON                       stepsizes and higher orders of extrapolation.
    a(1,i)=(func(x+hh)-func(x-hh))/(2.0*hh)       Try new, smaller stepsize.
    fac=CON2
    do 11 j=2,i                 Compute extrapolations of various orders, requiring no new
        a(j,i)=(a(j-1,i)*fac-a(j-1,i-1))/(fac-1.)          function evaluations.
        fac=CON2*fac
        errt=max(abs(a(j,i)-a(j-1,i)),abs(a(j,i)-a(j-1,i-1)))
          The error strategy is to compare each new extrapolation to one order lower, both at
          the present stepsize and the previous one.
        if (errt.le.err) then           If error is decreased, save the improved answer.
            err=errt
            dfridr=a(j,i)
        endif
    enddo 11
    if(abs(a(i,i)-a(i-1,i-1)).ge.SAFE*err)return
      If higher order is worse by a significant factor SAFE, then quit early.
enddo 12
return
END
```

In dfridr, the number of evaluations of func is typically 6 to 12, but is allowed to be as great as 2×NTAB. As a function of input h, it is typical for the accuracy to get *better* as h is made larger, until a sudden point is reached where nonsensical extrapolation produces early return with a large error. You should therefore choose a fairly large value for h, but monitor the returned value err, decreasing h if it is not small. For functions whose characteristic x scale is of order unity, we typically take h to be a few tenths.

Besides Ridders' method, there are other possible techniques. If your function is fairly smooth, and you know that you will want to evaluate its derivative many times at arbitrary points in some interval, then it makes sense to construct a Chebyshev polynomial approximation to the function in that interval, and to evaluate the derivative directly from the resulting Chebyshev coefficients. This method is described in §§5.8–5.9, following.

Another technique applies when the function consists of data that is tabulated at equally spaced intervals, and perhaps also noisy. One might then want, at each point, to least-squares *fit* a polynomial of some degree M, using an additional number n_L of points to the left and some number n_R of points to the right of each desired x value. The estimated derivative is then the derivative of the resulting fitted polynomial. A very efficient way to do this construction is via Savitzky-Golay smoothing filters, which will be discussed later, in §14.8. There we will give a routine for getting filter coefficients that not only construct the fitting polynomial but, in the accumulation of a single sum of data points times filter coefficients, evaluate it as well. In fact, the routine given, savgol, has an argument ld that determines which derivative of the fitted polynomial is evaluated. For the first derivative, the

appropriate setting is `ld=1`, and the value of the derivative is the accumulated sum divided by the sampling interval h.

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall), §§5.4–5.6. [1]

Ridders, C.J.F. 1982, *Advances in Engineering Software*, vol. 4, no. 2, pp. 75–76. [2]

5.8 Chebyshev Approximation

The Chebyshev polynomial of degree n is denoted $T_n(x)$, and is given by the explicit formula

$$T_n(x) = \cos(n \arccos x) \tag{5.8.1}$$

This may look trigonometric at first glance (and there is in fact a close relation between the Chebyshev polynomials and the discrete Fourier transform); however (5.8.1) can be combined with trigonometric identities to yield explicit expressions for $T_n(x)$ (see Figure 5.8.1),

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned} \tag{5.8.2}$$

(There also exist inverse formulas for the powers of x in terms of the T_n's — see equations 5.11.2-5.11.3.)

The Chebyshev polynomials are orthogonal in the interval $[-1, 1]$ over a weight $(1 - x^2)^{-1/2}$. In particular,

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases} \tag{5.8.3}$$

The polynomial $T_n(x)$ has n zeros in the interval $[-1, 1]$, and they are located at the points

$$x = \cos\left(\frac{\pi(k - \frac{1}{2})}{n}\right) \qquad k = 1, 2, \ldots, n \tag{5.8.4}$$

appropriate setting is `ld=1`, and the value of the derivative is the accumulated sum divided by the sampling interval h.

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall), §§5.4–5.6. [1]

Ridders, C.J.F. 1982, *Advances in Engineering Software*, vol. 4, no. 2, pp. 75–76. [2]

5.8 Chebyshev Approximation

The Chebyshev polynomial of degree n is denoted $T_n(x)$, and is given by the explicit formula

$$T_n(x) = \cos(n \arccos x) \tag{5.8.1}$$

This may look trigonometric at first glance (and there is in fact a close relation between the Chebyshev polynomials and the discrete Fourier transform); however (5.8.1) can be combined with trigonometric identities to yield explicit expressions for $T_n(x)$ (see Figure 5.8.1),

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned} \tag{5.8.2}$$

(There also exist inverse formulas for the powers of x in terms of the T_n's — see equations 5.11.2-5.11.3.)

The Chebyshev polynomials are orthogonal in the interval $[-1, 1]$ over a weight $(1 - x^2)^{-1/2}$. In particular,

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases} \tag{5.8.3}$$

The polynomial $T_n(x)$ has n zeros in the interval $[-1, 1]$, and they are located at the points

$$x = \cos\left(\frac{\pi(k - \frac{1}{2})}{n}\right) \qquad k = 1, 2, \ldots, n \tag{5.8.4}$$

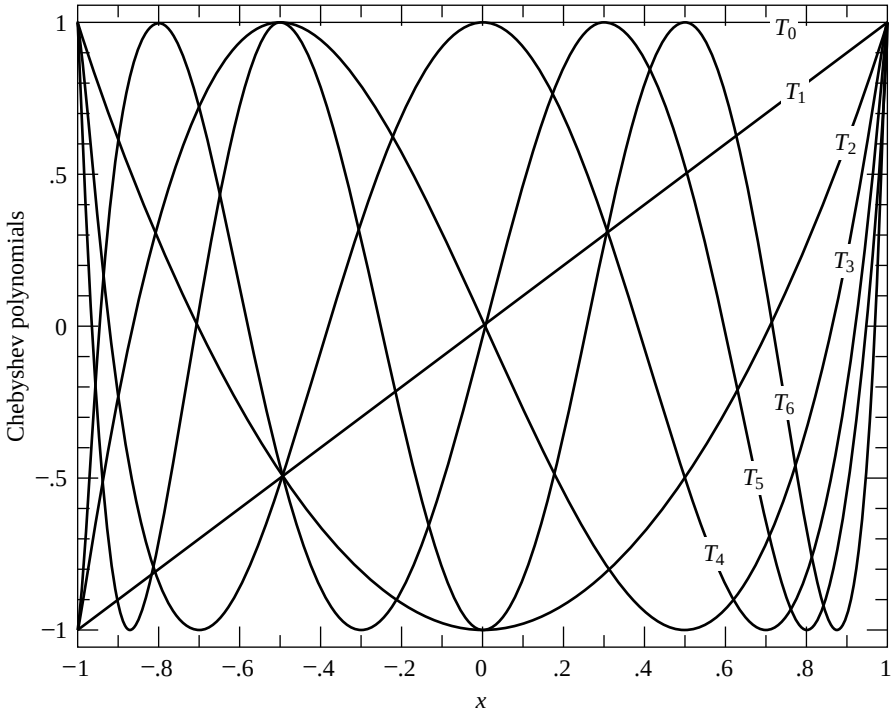


Figure 5.8.1. Chebyshev polynomials $T_0(x)$ through $T_6(x)$. Note that T_j has j roots in the interval $(-1, 1)$ and that all the polynomials are bounded between ± 1.

In this same interval there are $n + 1$ extrema (maxima and minima), located at

$$x = \cos\left(\frac{\pi k}{n}\right) \qquad k = 0, 1, \ldots, n \tag{5.8.5}$$

At all of the maxima $T_n(x) = 1$, while at all of the minima $T_n(x) = -1$; it is precisely this property that makes the Chebyshev polynomials so useful in polynomial approximation of functions.

The Chebyshev polynomials satisfy a discrete orthogonality relation as well as the continuous one (5.8.3): If x_k $(k = 1, \ldots, m)$ are the m zeros of $T_m(x)$ given by (5.8.4), and if $i, j < m$, then

$$\sum_{k=1}^{m} T_i(x_k) T_j(x_k) = \begin{cases} 0 & i \neq j \\ m/2 & i = j \neq 0 \\ m & i = j = 0 \end{cases} \tag{5.8.6}$$

It is not too difficult to combine equations (5.8.1), (5.8.4), and (5.8.6) to prove the following theorem: If $f(x)$ is an arbitrary function in the interval $[-1, 1]$, and if N coefficients $c_j, j = 1, \ldots, N$, are defined by

$$\begin{aligned} c_j &= \frac{2}{N} \sum_{k=1}^{N} f(x_k) T_{j-1}(x_k) \\ &= \frac{2}{N} \sum_{k=1}^{N} f\left[\cos\left(\frac{\pi(k - \frac{1}{2})}{N}\right)\right] \cos\left(\frac{\pi(j-1)(k-\frac{1}{2})}{N}\right) \end{aligned} \tag{5.8.7}$$

then the approximation formula

$$f(x) \approx \left[\sum_{k=1}^{N} c_k T_{k-1}(x) \right] - \frac{1}{2} c_1 \tag{5.8.8}$$

is *exact* for x equal to all of the N zeros of $T_N(x)$.

For a fixed N, equation (5.8.8) is a polynomial in x which approximates the function $f(x)$ in the interval $[-1, 1]$ (where all the zeros of $T_N(x)$ are located). Why is this particular approximating polynomial better than any other one, exact on some other set of N points? The answer is *not* that (5.8.8) is necessarily more accurate than some other approximating polynomial of the same order N (for some specified definition of "accurate"), but rather that (5.8.8) can be truncated to a polynomial of *lower* degree $m \ll N$ in a very graceful way, one that *does* yield the "most accurate" approximation of degree m (in a sense that can be made precise). Suppose N is so large that (5.8.8) is virtually a perfect approximation of $f(x)$. Now consider the truncated approximation

$$f(x) \approx \left[\sum_{k=1}^{m} c_k T_{k-1}(x) \right] - \frac{1}{2} c_1 \tag{5.8.9}$$

with the same c_j's, computed from (5.8.7). Since the $T_k(x)$'s are all bounded between ± 1, the difference between (5.8.9) and (5.8.8) can be no larger than the sum of the neglected c_k's ($k = m+1, \ldots, N$). In fact, if the c_k's are rapidly decreasing (which is the typical case), then the error is dominated by $c_{m+1} T_m(x)$, an oscillatory function with $m+1$ equal extrema distributed smoothly over the interval $[-1, 1]$. This smooth spreading out of the error is a very important property: The Chebyshev approximation (5.8.9) is very nearly the same polynomial as that holy grail of approximating polynomials the *minimax polynomial*, which (among all polynomials of the same degree) has the smallest maximum deviation from the true function $f(x)$. The minimax polynomial is very difficult to find; the Chebyshev approximating polynomial is almost identical and is very easy to compute!

So, given some (perhaps difficult) means of computing the function $f(x)$, we now need algorithms for implementing (5.8.7) and (after inspection of the resulting c_k's and choice of a truncating value m) evaluating (5.8.9). The latter equation then becomes an easy way of computing $f(x)$ for all subsequent time.

The first of these tasks is straightforward. A generalization of equation (5.8.7) that is here implemented is to allow the range of approximation to be between two arbitrary limits a and b, instead of just -1 to 1. This is effected by a change of variable

$$y \equiv \frac{x - \frac{1}{2}(b+a)}{\frac{1}{2}(b-a)} \tag{5.8.10}$$

and by the approximation of $f(x)$ by a Chebyshev polynomial in y.

```
SUBROUTINE chebft(a,b,c,n,func)
INTEGER n,NMAX
REAL a,b,c(n),func
DOUBLE PRECISION PI
EXTERNAL func
PARAMETER (NMAX=50, PI=3.141592653589793d0)
```

Chebyshev fit: Given a function `func`, lower and upper limits of the interval [a,b], and a maximum degree `n`, this routine computes the `n` coefficients c_k such that `func`$(x) \approx$

$[\sum_{k=1}^{\mathtt{n}} \mathtt{c}_k T_{k-1}(y)] - c_1/2$, where y and x are related by (5.8.10). This routine is to be used with moderately large n (e.g., 30 or 50), the array of c's subsequently to be truncated at the smaller value m such that $\mathtt{c}_{m+1}$ and subsequent elements are negligible.
Parameters: Maximum expected value of n, and π.

```
INTEGER j,k
REAL bma,bpa,fac,y,f(NMAX)
DOUBLE PRECISION sum
bma=0.5*(b-a)
bpa=0.5*(b+a)
do 11 k=1,n                          We evaluate the function at the n points required by (5.8.7).
    y=cos(PI*(k-0.5d0)/n)
    f(k)=func(y*bma+bpa)
enddo 11
fac=2./n
do 13 j=1,n
    sum=0.d0                         We will accumulate the sum in double precision, a nicety that
    do 12 k=1,n                          you can ignore.
        sum=sum+f(k)*cos((PI*(j-1))*((k-0.5d0)/n))
    enddo 12
    c(j)=fac*sum
enddo 13
return
END
```

(If you find that the execution time of chebft is dominated by the calculation of N^2 cosines, rather than by the N evaluations of your function, then you should look ahead to §12.3, especially equation 12.3.22, which shows how fast cosine transform methods can be used to evaluate equation 5.8.7.)

Now that we have the Chebyshev coefficients, how do we evaluate the approximation? One could use the recurrence relation of equation (5.8.2) to generate values for $T_k(x)$ from $T_0 = 1, T_1 = x$, while also accumulating the sum of (5.8.9). It is better to use Clenshaw's recurrence formula (§5.5), effecting the two processes simultaneously. Applied to the Chebyshev series (5.8.9), the recurrence is

$$\begin{aligned} d_{m+2} &\equiv d_{m+1} \equiv 0 \\ d_j &= 2x d_{j+1} - d_{j+2} + c_j \qquad j = m, m-1, \ldots, 2 \\ f(x) &\equiv d_0 = x d_2 - d_3 + \frac{1}{2} c_1 \end{aligned} \tag{5.8.11}$$

```
FUNCTION chebev(a,b,c,m,x)
INTEGER m
REAL chebev,a,b,x,c(m)
    Chebyshev evaluation: All arguments are input. c(1:m) is an array of Chebyshev coefficients, the first m elements of c output from chebft (which must have been called with the same a and b). The Chebyshev polynomial $\sum_{k=1}^{\mathtt{m}} \mathtt{c}_k T_{k-1}(y) - \mathtt{c}_1/2$ is evaluated at a point $\mathtt{y} = [\mathtt{x} - (\mathtt{b} + \mathtt{a})/2]/[(\mathtt{b} - \mathtt{a})/2]$, and the result is returned as the function value.
INTEGER j
REAL d,dd,sv,y,y2
if ((x-a)*(x-b).gt.0.) pause 'x not in range in chebev'
d=0.
dd=0.
y=(2.*x-a-b)/(b-a)                   Change of variable.
y2=2.*y
do 11 j=m,2,-1                       Clenshaw's recurrence.
    sv=d
```

```
    d=y2*d-dd+c(j)
    dd=sv
enddo 11
chebev=y*d-dd+0.5*c(1)        Last step is different.
return
END
```

If we are approximating an *even* function on the interval $[-1, 1]$, its expansion will involve only even Chebyshev polynomials. It is wasteful to call `chebev` with all the odd coefficients zero [1]. Instead, using the half-angle identity for the cosine in equation (5.8.1), we get the relation

$$T_{2n}(x) = T_n(2x^2 - 1) \tag{5.8.12}$$

Thus we can evaluate a series of even Chebyshev polynomials by calling `chebev` with the even coefficients stored consecutively in the array `c`, but with the argument x replaced by $2x^2 - 1$.

An odd function will have an expansion involving only odd Chebysev polynomials. It is best to rewrite it as an expansion for the function $f(x)/x$, which involves only even Chebyshev polynomials. This will give accurate values for $f(x)/x$ near $x = 0$. The coefficients c'_n for $f(x)/x$ can be found from those for $f(x)$ by recurrence:

$$\begin{aligned} c'_{N+1} &= 0 \\ c'_{n-1} &= 2c_n - c'_{n+1}, \qquad n = N, N-2, \ldots \end{aligned} \tag{5.8.13}$$

Equation (5.8.13) follows from the recurrence relation in equation (5.8.2).

If you insist on evaluating an odd Chebyshev series, the efficient way is to once again use `chebev` with x replaced by $y = 2x^2 - 1$, and with the odd coefficients stored consecutively in the array `c`. Now, however, you must also change the last formula in equation (5.8.11) to be

$$f(x) = x[(2y - 1)d_2 - d_3 + c_1] \tag{5.8.14}$$

and change the corresponding line in `chebev`.

CITED REFERENCES AND FURTHER READING:

Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory, (London: H.M. Stationery Office). [1]

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 8.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §4.4.1, p. 104.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.2, p. 334.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §1.10, p. 39.

5.9 Derivatives or Integrals of a Chebyshev-approximated Function

If you have obtained the Chebyshev coefficients that approximate a function in a certain range (e.g., from `chebft` in §5.8), then it is a simple matter to transform them to Chebyshev coefficients corresponding to the derivative or integral of the function. Having done this, you can evaluate the derivative or integral just as if it were a function that you had Chebyshev-fitted *ab initio*.

The relevant formulas are these: If $c_i,\ i = 1, \ldots, m$ are the coefficients that approximate a function f in equation (5.8.9), C_i are the coefficients that approximate the indefinite integral of f, and c'_i are the coefficients that approximate the derivative of f, then

$$C_i = \frac{c_{i-1} - c_{i+1}}{2(i-1)} \qquad (i > 1) \tag{5.9.1}$$

$$c'_{i-1} = c'_{i+1} + 2(i-1)c_i \qquad (i = m-1, m-2, \ldots, 2) \tag{5.9.2}$$

Equation (5.9.1) is augmented by an arbitrary choice of C_1, corresponding to an arbitrary constant of integration. Equation (5.9.2), which is a recurrence, is started with the values $c'_m = c'_{m+1} = 0$, corresponding to no information about the $m+1$st Chebyshev coefficient of the original function f.

Here are routines for implementing equations (5.9.1) and (5.9.2).

```
SUBROUTINE chder(a,b,c,cder,n)
INTEGER n
REAL a,b,c(n),cder(n)
    Given a,b,c(1:n), as output from routine chebft §5.8, and given n, the desired degree
    of approximation (length of c to be used), this routine returns the array cder(1:n), the
    Chebyshev coefficients of the derivative of the function whose coefficients are c(1:n).
INTEGER j
REAL con
cder(n)=0.                                  n and n-1 are special cases.
cder(n-1)=2*(n-1)*c(n)
do 11 j=n-2,1,-1
    cder(j)=cder(j+2)+2*j*c(j+1)            Equation (5.9.2).
enddo 11
con=2./(b-a)
do 12 j=1,n                                 Normalize to the interval b-a.
    cder(j)=cder(j)*con
enddo 12
return
END
```

```
SUBROUTINE chint(a,b,c,cint,n)
INTEGER n
REAL a,b,c(n),cint(n)
    Given a,b,c(1:n), as output from routine chebft §5.8, and given n, the desired degree
    of approximation (length of c to be used), this routine returns the array cint(1:n), the
    Chebyshev coefficients of the integral of the function whose coefficients are c. The constant
    of integration is set so that the integral vanishes at a.
INTEGER j
REAL con,fac,sum
```

5.9 Derivatives or Integrals of a Chebyshev-approximated Function

If you have obtained the Chebyshev coefficients that approximate a function in a certain range (e.g., from `chebft` in §5.8), then it is a simple matter to transform them to Chebyshev coefficients corresponding to the derivative or integral of the function. Having done this, you can evaluate the derivative or integral just as if it were a function that you had Chebyshev-fitted *ab initio*.

The relevant formulas are these: If $c_i,\ i = 1, \ldots, m$ are the coefficients that approximate a function f in equation (5.8.9), C_i are the coefficients that approximate the indefinite integral of f, and c'_i are the coefficients that approximate the derivative of f, then

$$C_i = \frac{c_{i-1} - c_{i+1}}{2(i-1)} \qquad (i > 1) \tag{5.9.1}$$

$$c'_{i-1} = c'_{i+1} + 2(i-1)c_i \qquad (i = m-1, m-2, \ldots, 2) \tag{5.9.2}$$

Equation (5.9.1) is augmented by an arbitrary choice of C_1, corresponding to an arbitrary constant of integration. Equation (5.9.2), which is a recurrence, is started with the values $c'_m = c'_{m+1} = 0$, corresponding to no information about the $m+1$st Chebyshev coefficient of the original function f.

Here are routines for implementing equations (5.9.1) and (5.9.2).

```
SUBROUTINE chder(a,b,c,cder,n)
INTEGER n
REAL a,b,c(n),cder(n)
    Given a,b,c(1:n), as output from routine chebft §5.8, and given n, the desired degree
    of approximation (length of c to be used), this routine returns the array cder(1:n), the
    Chebyshev coefficients of the derivative of the function whose coefficients are c(1:n).
INTEGER j
REAL con
cder(n)=0.                                  n and n-1 are special cases.
cder(n-1)=2*(n-1)*c(n)
do 11 j=n-2,1,-1
    cder(j)=cder(j+2)+2*j*c(j+1)            Equation (5.9.2).
enddo 11
con=2./(b-a)
do 12 j=1,n                                 Normalize to the interval b-a.
    cder(j)=cder(j)*con
enddo 12
return
END
```

```
SUBROUTINE chint(a,b,c,cint,n)
INTEGER n
REAL a,b,c(n),cint(n)
    Given a,b,c(1:n), as output from routine chebft §5.8, and given n, the desired degree
    of approximation (length of c to be used), this routine returns the array cint(1:n), the
    Chebyshev coefficients of the integral of the function whose coefficients are c. The constant
    of integration is set so that the integral vanishes at a.
INTEGER j
REAL con,fac,sum
```

```
con=0.25*(b-a)                  Factor that normalizes to the interval b-a.
sum=0.                          Accumulates the constant of integration.
fac=1.                          Will equal ±1.
do 11 j=2,n-1
    cint(j)=con*(c(j-1)-c(j+1))/(j-1)        Equation (5.9.1).
    sum=sum+fac*cint(j)
    fac=-fac
enddo 11
cint(n)=con*c(n-1)/(n-1)        Special case of (5.9.1) for n.
sum=sum+fac*cint(n)
cint(1)=2.*sum                  Set the constant of integration.
return
END
```

Clenshaw-Curtis Quadrature

Since a smooth function's Chebyshev coefficients c_i decrease rapidly, generally exponentially, equation (5.9.1) is often quite efficient as the basis for a quadrature scheme. The routines `chebft` and `chint`, used in that order, can be followed by repeated calls to `chebev` if $\int_a^x f(x)dx$ is required for many different values of x in the range $a \le x \le b$.

If only the single definite integral $\int_a^b f(x)dx$ is required, then `chint` and `chebev` are replaced by the simpler formula, derived from equation (5.9.1),

$$\int_a^b f(x)dx = (b-a)\left[\frac{1}{2}c_1 - \frac{1}{3}c_3 - \frac{1}{15}c_5 - \cdots - \frac{1}{(2k+1)(2k-1)}c_{2k+1} - \cdots\right] \tag{5.9.3}$$

where the c_i's are as returned by `chebft`. The series can be truncated when c_{2k+1} becomes negligible, and the first neglected term gives an error estimate.

This scheme is known as *Clenshaw-Curtis quadrature* [1]. It is often combined with an adaptive choice of N, the number of Chebyshev coefficients calculated via equation (5.8.7), which is also the number of function evaluations of $f(x)$. If a modest choice of N does not give a sufficiently small c_{2k+1} in equation (5.9.3), then a larger value is tried. In this adaptive case, it is even better to replace equation (5.8.7) by the so-called "trapezoidal" or Gauss-Lobatto (§4.5) variant,

$$c_j = \frac{2}{N}{\sum_{k=0}^{N}}'' f\left[\cos\left(\frac{\pi k}{N}\right)\right]\cos\left(\frac{\pi(j-1)k}{N}\right) \qquad j = 1, \ldots, N \tag{5.9.4}$$

where (N.B.!) the two primes signify that the first and last terms in the sum are to be multiplied by $1/2$. If N is doubled in equation (5.9.4), then half of the new function evaluation points are identical to the old ones, allowing the previous function evaluations to be reused. This feature, plus the analytic weights and abscissas (cosine functions in 5.9.4), give Clenshaw-Curtis quadrature an edge over high-order adaptive Gaussian quadrature (cf. §4.5), which the method otherwise resembles.

If your problem forces you to large values of N, you should be aware that equation (5.9.4) can be evaluated rapidly, and simultaneously for all the values of j, by a fast cosine transform. (See §12.3, especially equation 12.3.17.) (We already remarked that the nontrapezoidal form (5.8.7) can also be done by fast cosine methods, cf. equation 12.3.22.)

CITED REFERENCES AND FURTHER READING:

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), pp. 78–79.

Clenshaw, C.W., and Curtis, A.R. 1960, *Numerische Mathematik*, vol. 2, pp. 197–205. [1]

5.10 Polynomial Approximation from Chebyshev Coefficients

You may well ask after reading the preceding two sections, "Must I store and evaluate my Chebyshev approximation as an array of Chebyshev coefficients for a transformed variable y? Can't I convert the c_k's into actual polynomial coefficients in the original variable x and have an approximation of the following form?"

$$f(x) \approx \sum_{k=1}^{m} g_k x^{k-1} \tag{5.10.1}$$

Yes, you can do this (and we will give you the algorithm to do it), but we caution you against it: Evaluating equation (5.10.1), where the coefficient g's reflect an underlying Chebyshev approximation, usually requires more significant figures than evaluation of the Chebyshev sum directly (as by `chebev`). This is because the Chebyshev polynomials themselves exhibit a rather delicate cancellation: The leading coefficient of $T_n(x)$, for example, is 2^{n-1}; other coefficients of $T_n(x)$ are even bigger; yet they all manage to combine into a polynomial that lies between ± 1. *Only* when m is no larger than 7 or 8 should you contemplate writing a Chebyshev fit as a direct polynomial, and even in those cases you should be willing to tolerate two or so significant figures less accuracy than the roundoff limit of your machine.

You get the g's in equation (5.10.1) from the c's output from `chebft` (suitably truncated at a modest value of m) by calling in sequence the following two procedures:

```
SUBROUTINE chebpc(c,d,n)
INTEGER n,NMAX
REAL c(n),d(n)
PARAMETER (NMAX=50)           Maximum anticipated value of n.
    Chebyshev polynomial coefficients. Given a coefficient array c(1:n) of length n, this routine
    generates a coefficient array d(1:n) such that ∑_{k=1}^{n} d_k y^{k-1} = ∑_{k=1}^{n} c_k T_{k-1}(y) − c_1/2.
    The method is Clenshaw's recurrence (5.8.11), but now applied algebraically rather than
    arithmetically.
INTEGER j,k
REAL sv,dd(NMAX)
do 11 j=1,n
    d(j)=0.
    dd(j)=0.
enddo 11
d(1)=c(n)
do 13 j=n-1,2,-1
    do 12 k=n-j+1,2,-1
        sv=d(k)
        d(k)=2.*d(k-1)-dd(k)
        dd(k)=sv
    enddo 12
    sv=d(1)
    d(1)=-dd(1)+c(j)
    dd(1)=sv
enddo 13
do 14 j=n,2,-1
    d(j)=d(j-1)-dd(j)
enddo 14
d(1)=-dd(1)+0.5*c(1)
return
END
```

5.10 Polynomial Approximation from Chebyshev Coefficients

You may well ask after reading the preceding two sections, "Must I store and evaluate my Chebyshev approximation as an array of Chebyshev coefficients for a transformed variable y? Can't I convert the c_k's into actual polynomial coefficients in the original variable x and have an approximation of the following form?"

$$f(x) \approx \sum_{k=1}^{m} g_k x^{k-1} \tag{5.10.1}$$

Yes, you can do this (and we will give you the algorithm to do it), but we caution you against it: Evaluating equation (5.10.1), where the coefficient g's reflect an underlying Chebyshev approximation, usually requires more significant figures than evaluation of the Chebyshev sum directly (as by `chebev`). This is because the Chebyshev polynomials themselves exhibit a rather delicate cancellation: The leading coefficient of $T_n(x)$, for example, is 2^{n-1}; other coefficients of $T_n(x)$ are even bigger; yet they all manage to combine into a polynomial that lies between ± 1. *Only* when m is no larger than 7 or 8 should you contemplate writing a Chebyshev fit as a direct polynomial, and even in those cases you should be willing to tolerate two or so significant figures less accuracy than the roundoff limit of your machine.

You get the g's in equation (5.10.1) from the c's output from `chebft` (suitably truncated at a modest value of m) by calling in sequence the following two procedures:

```
SUBROUTINE chebpc(c,d,n)
INTEGER n,NMAX
REAL c(n),d(n)
PARAMETER (NMAX=50)           Maximum anticipated value of n.
    Chebyshev polynomial coefficients. Given a coefficient array c(1:n) of length n, this routine
    generates a coefficient array d(1:n) such that ∑_{k=1}^{n} d_k y^{k-1} = ∑_{k=1}^{n} c_k T_{k-1}(y) − c_1/2.
    The method is Clenshaw's recurrence (5.8.11), but now applied algebraically rather than
    arithmetically.
INTEGER j,k
REAL sv,dd(NMAX)
do 11 j=1,n
    d(j)=0.
    dd(j)=0.
enddo 11
d(1)=c(n)
do 13 j=n-1,2,-1
    do 12 k=n-j+1,2,-1
        sv=d(k)
        d(k)=2.*d(k-1)-dd(k)
        dd(k)=sv
    enddo 12
    sv=d(1)
    d(1)=-dd(1)+c(j)
    dd(1)=sv
enddo 13
do 14 j=n,2,-1
    d(j)=d(j-1)-dd(j)
enddo 14
d(1)=-dd(1)+0.5*c(1)
return
END
```

```
SUBROUTINE pcshft(a,b,d,n)
INTEGER n
REAL a,b,d(n)
    Polynomial coefficient shift. Given a coefficient array d(1:n), this routine generates a
    coefficient array g(1:n) such that sum_{k=1}^{n} d_k y^{k-1} = sum_{k=1}^{n} g_k x^{k-1}, where x and y are
    related by (5.8.10), i.e., the interval -1 < y < 1 is mapped to the interval a < x < b.
    The array g is returned in d.
INTEGER j,k
REAL const,fac
const=2./(b-a)
fac=const
do 11 j=2,n                          First we rescale by the factor const...
    d(j)=d(j)*fac
    fac=fac*const
enddo 11
const=0.5*(a+b)                      ...which is then redefined as the desired shift.
do 13 j=1,n-1                        We accomplish the shift by synthetic division. Synthetic
    do 12 k=n-1,j,-1                     division is a miracle of high-school algebra. If you
        d(k)=d(k)-const*d(k+1)           never learned it, go do so. You won't be sorry.
    enddo 12
enddo 13
return
END
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 59, 182–183 [synthetic division].

5.11 Economization of Power Series

One particular application of Chebyshev methods, the *economization of power series*, is an occasionally useful technique, with a flavor of getting something for nothing.

Suppose that you are already computing a function by the use of a convergent power series, for example

$$f(x) \equiv 1 - \frac{x}{3!} + \frac{x^2}{5!} - \frac{x^3}{7!} + \cdots \tag{5.11.1}$$

(This function is actually $\sin(\sqrt{x})/\sqrt{x}$, but pretend you don't know that.) You might be doing a problem that requires evaluating the series many times in some particular interval, say $[0, (2\pi)^2]$. Everything is fine, except that the series requires a large number of terms before its error (approximated by the first neglected term, say) is tolerable. In our example, with $x = (2\pi)^2$, the first term smaller than 10^{-7} is $x^{13}/(27!)$. This then approximates the error of the finite series whose last term is $x^{12}/(25!)$.

Notice that because of the large exponent in x^{13}, the error is *much smaller* than 10^{-7} everywhere in the interval except at the very largest values of x. This is the feature that allows "economization": if we are willing to let the error elsewhere in the interval rise to about the same value that the first neglected term has at the extreme end of the interval, then we can replace the 13-term series by one that is significantly shorter.

Here are the steps for doing so:

1. Change variables from x to y, as in equation (5.8.10), to map the x interval into $-1 \le y \le 1$.
2. Find the coefficients of the Chebyshev sum (like equation 5.8.8) that exactly equals your truncated power series (the one with enough terms for accuracy).

```
SUBROUTINE pcshft(a,b,d,n)
INTEGER n
REAL a,b,d(n)
    Polynomial coefficient shift. Given a coefficient array d(1:n), this routine generates a
    coefficient array g(1:n) such that sum_{k=1}^{n} d_k y^{k-1} = sum_{k=1}^{n} g_k x^{k-1}, where x and y are
    related by (5.8.10), i.e., the interval -1 < y < 1 is mapped to the interval a < x < b.
    The array g is returned in d.
INTEGER j,k
REAL const,fac
const=2./(b-a)
fac=const
do 11 j=2,n                          First we rescale by the factor const...
    d(j)=d(j)*fac
    fac=fac*const
enddo 11
const=0.5*(a+b)                      ...which is then redefined as the desired shift.
do 13 j=1,n-1                        We accomplish the shift by synthetic division. Synthetic
    do 12 k=n-1,j,-1                     division is a miracle of high-school algebra. If you
        d(k)=d(k)-const*d(k+1)           never learned it, go do so. You won't be sorry.
    enddo 12
enddo 13
return
END
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 59, 182–183 [synthetic division].

5.11 Economization of Power Series

One particular application of Chebyshev methods, the *economization of power series*, is an occasionally useful technique, with a flavor of getting something for nothing.

Suppose that you are already computing a function by the use of a convergent power series, for example

$$f(x) \equiv 1 - \frac{x}{3!} + \frac{x^2}{5!} - \frac{x^3}{7!} + \cdots \tag{5.11.1}$$

(This function is actually $\sin(\sqrt{x})/\sqrt{x}$, but pretend you don't know that.) You might be doing a problem that requires evaluating the series many times in some particular interval, say $[0, (2\pi)^2]$. Everything is fine, except that the series requires a large number of terms before its error (approximated by the first neglected term, say) is tolerable. In our example, with $x = (2\pi)^2$, the first term smaller than 10^{-7} is $x^{13}/(27!)$. This then approximates the error of the finite series whose last term is $x^{12}/(25!)$.

Notice that because of the large exponent in x^{13}, the error is *much smaller* than 10^{-7} everywhere in the interval except at the very largest values of x. This is the feature that allows "economization": if we are willing to let the error elsewhere in the interval rise to about the same value that the first neglected term has at the extreme end of the interval, then we can replace the 13-term series by one that is significantly shorter.

Here are the steps for doing so:

1. Change variables from x to y, as in equation (5.8.10), to map the x interval into $-1 \le y \le 1$.
2. Find the coefficients of the Chebyshev sum (like equation 5.8.8) that exactly equals your truncated power series (the one with enough terms for accuracy).

3. Truncate this Chebyshev series to a smaller number of terms, using the coefficient of the first neglected Chebyshev polynomial as an estimate of the error.
4. Convert back to a polynomial in y.
5. Change variables back to x.

All of these steps can be done numerically, given the coefficients of the original power series expansion. The first step is exactly the inverse of the routine `pcshft` (§5.10), which mapped a polynomial from y (in the interval $[-1,1]$) to x (in the interval $[a,b]$). But since equation (5.8.10) is a linear relation between x and y, one can also use `pcshft` for the inverse. The inverse of

$$\texttt{pcshft}(a,b,\texttt{d},\texttt{n})$$

turns out to be (you can check this)

$$\texttt{pcshft}\left(\frac{-2-b-a}{b-a},\frac{2-b-a}{b-a},\texttt{d},\texttt{n}\right)$$

The second step requires the inverse operation to that done by the routine `chebpc` (which took Chebyshev coefficients into polynomial coefficients). The following routine, `pccheb`, accomplishes this, using the formula [1]

$$x^k = \frac{1}{2^{k-1}}\left[T_k(x) + \binom{k}{1}T_{k-2}(x) + \binom{k}{2}T_{k-4}(x) + \cdots\right] \tag{5.11.2}$$

where the last term depends on whether k is even or odd,

$$\cdots + \binom{k}{(k-1)/2}T_1(x) \quad (k \text{ odd}), \qquad \cdots + \frac{1}{2}\binom{k}{k/2}T_0(x) \quad (k \text{ even}). \tag{5.11.3}$$

```
SUBROUTINE pccheb(d,c,n)
INTEGER n
REAL c(n),d(n)
    Inverse of routine chebpc: given an array of polynomial coefficients d(1:n), returns an
    equivalent array of Chebyshev coefficients c(1:n).
INTEGER j,jm,jp,k
REAL fac,pow
pow=1.                          Will be powers of 2.
c(1)=2.*d(1)
do 12 k=2,n                     Loop over orders of x in the polynomial.
    c(k)=0.                     Zero corresponding order of Chebyshev.
    fac=d(k)/pow
    jm=k-1
    jp=1
    do 11 j=k,1,-2              Increment this and lower orders of Chebyshev with the com-
        c(j)=c(j)+fac               binatorial coefficent times d(k); see text for formula.
        fac=fac*float(jm)/float(jp)
        jm=jm-1
        jp=jp+1
    enddo 11
    pow=2.*pow
enddo 12
return
END
```

The fourth and fifth steps are accomplished by the routines `chebpc` and `pcshft`, respectively. Here is how the procedure looks all together:

```
INTEGER NMANY,NFEW
REAL e(NMANY),d(NFEW),c(NMANY),a,b
```
Economize NMANY power series coefficients e(1:NMANY) in the range (a, b) into NFEW coefficients d(1:NFEW).
```
call pcshft((-2.-b-a)/(b-a),(2.-b-a)/(b-a),e,NMANY)
call pccheb(e,c,NMANY)
...
```
Here one would normally examine the Chebyshev coefficients c(1:NMANY) to decide how small NFEW can be.
```
call chebpc(c,d,NFEW)
call pcshft(a,b,d,NFEW)
```

In our example, by the way, the 8th through 10th Chebyshev coefficients turn out to be on the order of -7×10^{-6}, 3×10^{-7}, and -9×10^{-9}, so reasonable truncations (for single precision calculations) are somewhere in this range, yielding a polynomial with 8 – 10 terms instead of the original 13.

Replacing a 13-term polynomial with a (say) 10-term polynomial without any loss of accuracy — that does seem to be getting something for nothing. Is there some magic in this technique? Not really. The 13-term polynomial defined a function $f(x)$. Equivalent to economizing the series, we could instead have evaluated $f(x)$ at enough points to construct its Chebyshev approximation in the interval of interest, by the methods of §5.8. We would have obtained just the same lower-order polynomial. The principal lesson is that the rate of convergence of Chebyshev coefficients has nothing to do with the rate of convergence of power series coefficients; and it is the *former* that dictates the number of terms needed in a polynomial approximation. A function might have a *divergent* power series in some region of interest, but if the function itself is well-behaved, it will have perfectly good polynomial approximations. These can be found by the methods of §5.8, but *not* by economization of series. There is slightly less to economization of series than meets the eye.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 12.

Arfken, G. 1970, *Mathematical Methods for Physicists*, 2nd ed. (New York: Academic Press), p. 631. [1]

5.12 Padé Approximants

A *Padé approximant*, so called, is that rational function (of a specified order) whose power series expansion agrees with a given power series to the highest possible order. If the rational function is

$$R(x) \equiv \frac{\sum_{k=0}^{M} a_k x^k}{1 + \sum_{k=1}^{N} b_k x^k} \tag{5.12.1}$$

then $R(x)$ is said to be a Padé approximant to the series

$$f(x) \equiv \sum_{k=0}^{\infty} c_k x^k \tag{5.12.2}$$

if

$$R(0) = f(0) \tag{5.12.3}$$

```
INTEGER NMANY,NFEW
REAL e(NMANY),d(NFEW),c(NMANY),a,b
```
Economize NMANY power series coefficients e(1:NMANY) in the range (a, b) into NFEW coefficients d(1:NFEW).
```
call pcshft((-2.-b-a)/(b-a),(2.-b-a)/(b-a),e,NMANY)
call pccheb(e,c,NMANY)
...
```
Here one would normally examine the Chebyshev coefficients c(1:NMANY) to decide how small NFEW can be.
```
call chebpc(c,d,NFEW)
call pcshft(a,b,d,NFEW)
```

In our example, by the way, the 8th through 10th Chebyshev coefficients turn out to be on the order of -7×10^{-6}, 3×10^{-7}, and -9×10^{-9}, so reasonable truncations (for single precision calculations) are somewhere in this range, yielding a polynomial with 8 – 10 terms instead of the original 13.

Replacing a 13-term polynomial with a (say) 10-term polynomial without any loss of accuracy — that does seem to be getting something for nothing. Is there some magic in this technique? Not really. The 13-term polynomial defined a function $f(x)$. Equivalent to economizing the series, we could instead have evaluated $f(x)$ at enough points to construct its Chebyshev approximation in the interval of interest, by the methods of §5.8. We would have obtained just the same lower-order polynomial. The principal lesson is that the rate of convergence of Chebyshev coefficients has nothing to do with the rate of convergence of power series coefficients; and it is the *former* that dictates the number of terms needed in a polynomial approximation. A function might have a *divergent* power series in some region of interest, but if the function itself is well-behaved, it will have perfectly good polynomial approximations. These can be found by the methods of §5.8, but *not* by economization of series. There is slightly less to economization of series than meets the eye.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 12.

Arfken, G. 1970, *Mathematical Methods for Physicists*, 2nd ed. (New York: Academic Press), p. 631. [1]

5.12 Padé Approximants

A *Padé approximant*, so called, is that rational function (of a specified order) whose power series expansion agrees with a given power series to the highest possible order. If the rational function is

$$R(x) \equiv \frac{\sum_{k=0}^{M} a_k x^k}{1 + \sum_{k=1}^{N} b_k x^k} \tag{5.12.1}$$

then $R(x)$ is said to be a Padé approximant to the series

$$f(x) \equiv \sum_{k=0}^{\infty} c_k x^k \tag{5.12.2}$$

if

$$R(0) = f(0) \tag{5.12.3}$$

and also

$$\left.\frac{d^k}{dx^k}R(x)\right|_{x=0} = \left.\frac{d^k}{dx^k}f(x)\right|_{x=0}, \qquad k = 1, 2, \ldots, M+N \tag{5.12.4}$$

Equations (5.12.3) and (5.12.4) furnish $M + N + 1$ equations for the unknowns $a_0, \ldots, a_M$ and $b_1, \ldots, b_N$. The easiest way to see what these equations are is to equate (5.12.1) and (5.12.2), multiply both by the denominator of equation (5.12.1), and equate all powers of x that have either a's or b's in their coefficients. If we consider only the special case of a diagonal rational approximation, $M = N$ (cf. §3.2), then we have $a_0 = c_0$, with the remaining a's and b's satisfying

$$\sum_{m=1}^{N} b_m c_{N-m+k} = -c_{N+k}, \qquad k = 1, \ldots, N \tag{5.12.5}$$

$$\sum_{m=0}^{k} b_m c_{k-m} = a_k, \qquad k = 1, \ldots, N \tag{5.12.6}$$

(note, in equation 5.12.1, that $b_0 = 1$). To solve these, start with equations (5.12.5), which are a set of linear equations for all the unknown b's. Although the set is in the form of a Toeplitz matrix (compare equation 2.8.8), experience shows that the equations are frequently close to singular, so that one should not solve them by the methods of §2.8, but rather by full LU decomposition. Additionally, it is a good idea to refine the solution by iterative improvement (routine `mprove` in §2.5) [1].

Once the b's are known, then equation (5.12.6) gives an explicit formula for the unknown a's, completing the solution.

Padé approximants are typically used when there is some unknown underlying function $f(x)$. We suppose that you are able somehow to compute, perhaps by laborious analytic expansions, the values of $f(x)$ and a few of its derivatives at $x = 0$: $f(0)$, $f'(0)$, $f''(0)$, and so on. These are of course the first few coefficients in the power series expansion of $f(x)$; but they are not necessarily getting small, and you have no idea where (or whether) the power series is convergent.

By contrast with techniques like Chebyshev approximation (§5.8) or economization of power series (§5.11) that only condense the information that you already know about a function, Padé approximants can give you genuinely new information about your function's values. It is sometimes quite mysterious how well this can work. (Like other mysteries in mathematics, it relates to *analyticity*.) An example will illustrate.

Imagine that, by extraordinary labors, you have ground out the first five terms in the power series expansion of an unknown function $f(x)$,

$$f(x) \approx 2 + \frac{1}{9}x + \frac{1}{81}x^2 - \frac{49}{8748}x^3 + \frac{175}{78732}x^4 + \cdots \tag{5.12.7}$$

(It is not really necessary that you know the coefficients in exact rational form — numerical values are just as good. We here write them as rationals to give you the impression that they derive from some side analytic calculation.) Equation (5.12.7) is plotted as the curve labeled "power series" in Figure 5.12.1. One sees that for $x \gtrsim 4$ it is dominated by its largest, quartic, term.

We now take the five coefficients in equation (5.12.7) and run them through the routine `pade` listed below. It returns five rational coefficients, three a's and two b's, for use in equation (5.12.1) with $M = N = 2$. The curve in the figure labeled "Padé" plots the resulting rational function. Note that both solid curves derive from the *same* five original coefficient values.

To evaluate the results, we need *Deus ex machina* (a useful fellow, when he is available) to tell us that equation (5.12.7) is in fact the power series expansion of the function

$$f(x) = [7 + (1 + x)^{4/3}]^{1/3} \tag{5.12.8}$$

which is plotted as the dotted curve in the figure. This function has a branch point at $x = -1$, so its power series is convergent only in the range $-1 < x < 1$. In most of the range shown in the figure, the series is divergent, and the value of its truncation to five terms is

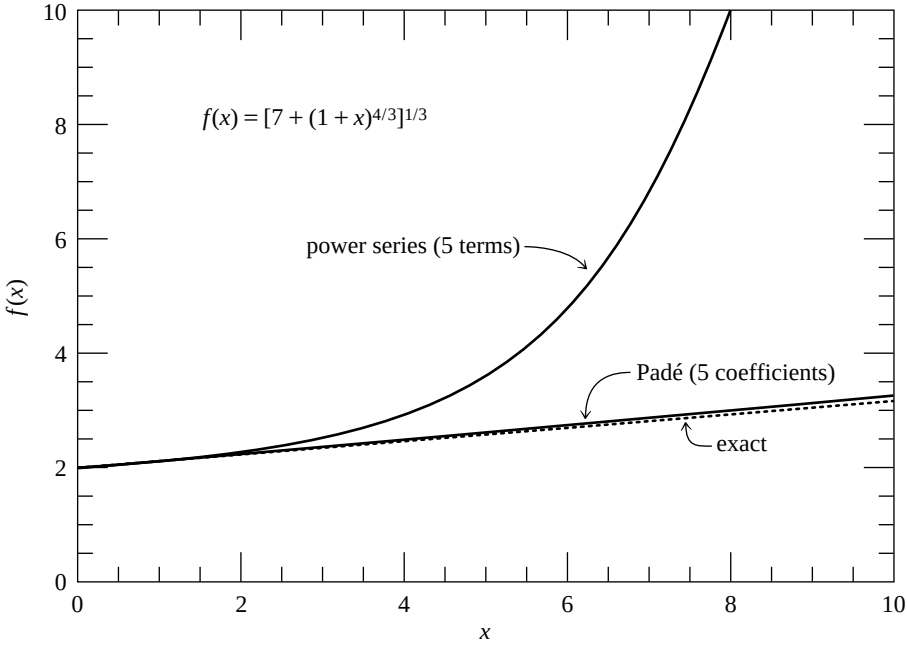


Figure 5.12.1. The five-term power series expansion and the derived five-coefficient Padé approximant for a sample function $f(x)$. The full power series converges only for $x < 1$. Note that the Padé approximant maintains accuracy far outside the radius of convergence of the series.

rather meaningless. Nevertheless, those five terms, converted to a Padé approximant, give a remarkably good representation of the function up to at least $x \sim 10$.

Why does this work? Are there not other functions with the same first five terms in their power series, but completely different behavior in the range (say) $2 < x < 10$? Indeed there are. Padé approximation has the uncanny knack of picking the function *you had in mind* from among all the possibilities. *Except when it doesn't!* That is the downside of Padé approximation: it is uncontrolled. There is, in general, no way to tell how accurate it is, or how far out in x it can usefully be extended. It is a powerful, but in the end still mysterious, technique.

Here is the routine that gets a's and b's from your c's. Note that the routine is specialized to the case $M = N$, and also that, on output, the rational coefficients are arranged in a format for use with the evaluation routine `ratval` (§5.3). (Also for consistency with that routine, the array of c's is passed in double precision.)

```
      SUBROUTINE pade(cof,n,resid)
      INTEGER n,NMAX
      REAL resid,BIG
      DOUBLE PRECISION cof(2*n+1)          For consistency with ratval.
      PARAMETER (NMAX=20,BIG=1.E30)        Max expected value of n, and a big number.
C     USES lubksb,ludcmp,mprove
        Given cof(1:2*n+1), the leading terms in the power series expansion of a function, solve
        the linear Padé equations to return the coefficients of a diagonal rational function approx-
        imation to the same function, namely (cof(1) + cof(2)x + ··· + cof(n+1)x^N)/(1 +
        cof(n+2)x + ··· + cof(2*n+1)x^N). The value resid is the norm of the residual vector;
        a small value indicates a well-converged solution.
      INTEGER j,k,indx(NMAX)
      REAL d,rr,rrold,sum,q(NMAX,NMAX),qlu(NMAX,NMAX),x(NMAX),
     *     y(NMAX),z(NMAX)
      do 12 j=1,n                          Set up matrix for solving.
        x(j)=cof(n+j+1)
```

```
        y(j)=x(j)
        do 11 k=1,n
            q(j,k)=cof(j-k+n+1)
            qlu(j,k)=q(j,k)
        enddo 11
    enddo 12
    call ludcmp(qlu,n,NMAX,indx,d)      Solve by LU decomposition and backsubstitution.
    call lubksb(qlu,n,NMAX,indx,x)
    rr=BIG
1   continue                            Important to use iterative improvement, since the
        rrold=rr                            Padé equations tend to be ill-conditioned.
        do 13 j=1,n
            z(j)=x(j)
        enddo 13
        call mprove(q,qlu,n,NMAX,indx,y,x)
        rr=0.
        do 14 j=1,n                     Calculate residual.
            rr=rr+(z(j)-x(j))**2
        enddo 14
    if(rr.lt.rrold)goto 1               If it is no longer improving, call it quits.
    resid=sqrt(rr)
    do 16 k=1,n                         Calculate the remaining coefficients.
        sum=cof(k+1)
        do 15 j=1,k
            sum=sum-x(j)*cof(k-j+1)
        enddo 15
        y(k)=sum
    enddo 16                            Copy answers to output.
    do 17 j=1,n
        cof(j+1)=y(j)
        cof(j+n+1)=-x(j)
    enddo 17
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), p. 14.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 2.

Graves-Morris, P.R. 1979, in *Padé Approximation and Its Applications*, Lecture Notes in Mathematics, vol. 765, L. Wuytack, ed. (Berlin: Springer-Verlag). [1]

5.13 Rational Chebyshev Approximation

In §5.8 and §5.10 we learned how to find good polynomial approximations to a given function $f(x)$ in a given interval $a \leq x \leq b$. Here, we want to generalize the task to find good approximations that are rational functions (see §5.3). The reason for doing so is that, for some functions and some intervals, the optimal rational function approximation is able to achieve substantially higher accuracy than the optimal polynomial approximation with the same number of coefficients. This must be weighed against the fact that finding a rational function approximation is not as straightforward as finding a polynomial approximation, which, as we saw, could be done elegantly via Chebyshev polynomials.

```
        y(j)=x(j)
        do 11 k=1,n
            q(j,k)=cof(j-k+n+1)
            qlu(j,k)=q(j,k)
        enddo 11
    enddo 12
    call ludcmp(qlu,n,NMAX,indx,d)      Solve by LU decomposition and backsubstitution.
    call lubksb(qlu,n,NMAX,indx,x)
    rr=BIG
1   continue                            Important to use iterative improvement, since the
        rrold=rr                            Padé equations tend to be ill-conditioned.
        do 13 j=1,n
            z(j)=x(j)
        enddo 13
        call mprove(q,qlu,n,NMAX,indx,y,x)
        rr=0.
        do 14 j=1,n                     Calculate residual.
            rr=rr+(z(j)-x(j))**2
        enddo 14
    if(rr.lt.rrold)goto 1               If it is no longer improving, call it quits.
    resid=sqrt(rr)
    do 16 k=1,n                         Calculate the remaining coefficients.
        sum=cof(k+1)
        do 15 j=1,k
            sum=sum-x(j)*cof(k-j+1)
        enddo 15
        y(k)=sum
    enddo 16                            Copy answers to output.
    do 17 j=1,n
        cof(j+1)=y(j)
        cof(j+n+1)=-x(j)
    enddo 17
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), p. 14.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 2.

Graves-Morris, P.R. 1979, in *Padé Approximation and Its Applications*, Lecture Notes in Mathematics, vol. 765, L. Wuytack, ed. (Berlin: Springer-Verlag). [1]

5.13 Rational Chebyshev Approximation

In §5.8 and §5.10 we learned how to find good polynomial approximations to a given function $f(x)$ in a given interval $a \leq x \leq b$. Here, we want to generalize the task to find good approximations that are rational functions (see §5.3). The reason for doing so is that, for some functions and some intervals, the optimal rational function approximation is able to achieve substantially higher accuracy than the optimal polynomial approximation with the same number of coefficients. This must be weighed against the fact that finding a rational function approximation is not as straightforward as finding a polynomial approximation, which, as we saw, could be done elegantly via Chebyshev polynomials.

Let the desired rational function $R(x)$ have numerator of degree m and denominator of degree k. Then we have

$$R(x) \equiv \frac{p_0 + p_1 x + \cdots + p_m x^m}{1 + q_1 x + \cdots + q_k x^k} \approx f(x) \qquad \text{for } a \le x \le b \tag{5.13.1}$$

The unknown quantities that we need to find are $p_0, \ldots, p_m$ and $q_1, \ldots, q_k$, that is, $m + k + 1$ quantities in all. Let $r(x)$ denote the deviation of $R(x)$ from $f(x)$, and let r denote its maximum absolute value,

$$r(x) \equiv R(x) - f(x) \qquad r \equiv \max_{a \le x \le b} |r(x)| \tag{5.13.2}$$

The ideal *minimax* solution would be that choice of p's and q's that minimizes r. Obviously there is *some* minimax solution, since r is bounded below by zero. How can we find it, or a reasonable approximation to it?

A first hint is furnished by the following fundamental theorem: If $R(x)$ is nondegenerate (has no common polynomial factors in numerator and denominator), then there is a unique choice of p's and q's that minimizes r; for this choice, $r(x)$ has $m + k + 2$ extrema in $a \le x \le b$, *all of magnitude* r *and with alternating sign.* (We have omitted some technical assumptions in this theorem. See Ralston [1] for a precise statement.) We thus learn that the situation with rational functions is quite analogous to that for minimax polynomials: In §5.8 we saw that the error term of an nth order approximation, with $n + 1$ Chebyshev coefficients, was generally dominated by the first neglected Chebyshev term, namely T_{n+1}, which itself has $n + 2$ extrema of equal magnitude and alternating sign. So, here, the number of rational coefficients, $m + k + 1$, plays the same role of the number of polynomial coefficients, $n + 1$.

A different way to see why $r(x)$ should have $m + k + 2$ extrema is to note that $R(x)$ can be made exactly equal to $f(x)$ at any $m + k + 1$ points x_i. Multiplying equation (5.13.1) by its denominator gives the equations

$$\begin{aligned} p_0 + p_1 x_i + \cdots + p_m x_i^m &= f(x_i)(1 + q_1 x_i + \cdots + q_k x_i^k) \\ i &= 1, 2, \ldots, m + k + 1 \end{aligned} \tag{5.13.3}$$

This is a set of $m + k + 1$ linear equations for the unknown p's and q's, which can be solved by standard methods (e.g., LU decomposition). If we choose the x_i's to all be in the interval (a, b), then there will generically be an extremum between each chosen x_i and x_{i+1}, plus also extrema where the function goes out of the interval at a and b, for a total of $m + k + 2$ extrema. For arbitrary x_i's, the extrema will not have the same magnitude. The theorem says that, for one particular choice of x_i's, the magnitudes can be beaten down to the identical, minimal, value of r.

Instead of making $f(x_i)$ and $R(x_i)$ equal at the points x_i, one can instead force the residual $r(x_i)$ to any desired values y_i by solving the linear equations

$$\begin{aligned} p_0 + p_1 x_i + \cdots + p_m x_i^m &= [f(x_i) - y_i](1 + q_1 x_i + \cdots + q_k x_i^k) \\ i &= 1, 2, \ldots, m + k + 1 \end{aligned} \tag{5.13.4}$$

In fact, if the x_i's are chosen to be the extrema (not the zeros) of the minimax solution, then the equations satisfied will be

$$\begin{aligned} p_0 + p_1 x_i + \cdots + p_m x_i^m &= [f(x_i) \pm r](1 + q_1 x_i + \cdots + q_k x_i^k) \\ i &= 1, 2, \ldots, m + k + 2 \end{aligned} \tag{5.13.5}$$

where the $\pm$ alternates for the alternating extrema. Notice that equation (5.13.5) is satisfied at $m + k + 2$ extrema, while equation (5.13.4) was satisfied only at $m + k + 1$ arbitrary points. How can this be? The answer is that r in equation (5.13.5) is an additional unknown, so that the number of both equations and unknowns is $m + k + 2$. True, the set is mildly nonlinear (in r), but in general it is still perfectly soluble by methods that we will develop in Chapter 9.

We thus see that, given only the *locations* of the extrema of the minimax rational function, we can solve for its coefficients and maximum deviation. Additional theorems,

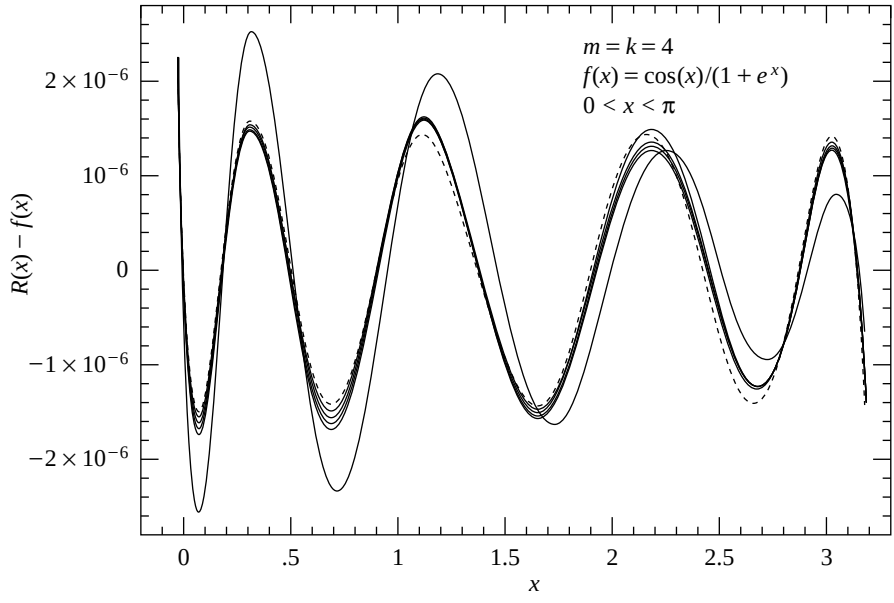


Figure 5.13.1. Solid curves show deviations $r(x)$ for five successive iterations of the routine `ratlsq` for an arbitrary test problem. The algorithm does not converge to exactly the minimax solution (shown as the dotted curve). But, after one iteration, the discrepancy is a small fraction of the last significant bit of accuracy.

leading up to the so-called *Remes algorithms* [1], tell how to converge to these locations by an iterative process. For example, here is a (slightly simplified) statement of *Remes' Second Algorithm*: (1) Find an initial rational function with $m + k + 2$ extrema x_i (not having equal deviation). (2) Solve equation (5.13.5) for new rational coefficients and r. (3) Evaluate the resulting $R(x)$ to find its actual extrema (which will not be the same as the guessed values). (4) Replace each guessed value with the nearest actual extremum of the same sign. (5) Go back to step 2 and iterate to convergence. Under a broad set of assumptions, this method will converge. Ralston [1] fills in the necessary details, including how to find the initial set of x_i's.

Up to this point, our discussion has been textbook-standard. We now reveal ourselves as heretics. We don't much like the elegant Remes algorithm. Its two nested iterations (on r in the nonlinear set 5.13.5, and on the new sets of x_i's) are finicky and require a lot of special logic for degenerate cases. Even more heretical, we doubt that compulsive searching for the *exactly best*, equal deviation, approximation is worth the effort — except perhaps for those few people in the world whose business it is to find optimal approximations that get built into compilers and microchips.

When we use rational function approximation, the goal is usually much more pragmatic: Inside some inner loop we are evaluating some function a zillion times, and we want to speed up its evaluation. Almost never do we need this function to the last bit of machine accuracy. Suppose (heresy!) we use an approximation whose error has $m + k + 2$ extrema whose deviations differ by a factor of 2. The theorems on which the Remes algorithms are based guarantee that the perfect minimax solution will have extrema somewhere within this factor of 2 range – forcing down the higher extrema will cause the lower ones to rise, until all are equal. So our "sloppy" approximation is in fact within a fraction of a least significant bit of the minimax one.

That is good enough for us, especially when we have available a very robust method for finding the so-called "sloppy" approximation. Such a method is the least-squares solution of overdetermined linear equations by singular value decomposition (§2.6 and §15.4). We proceed as follows: First, solve (in the least-squares sense) equation (5.13.3), not just for $m + k + 1$ values of x_i, but for a significantly larger number of x_i's, spaced approximately

like the zeros of a high-order Chebyshev polynomial. This gives an initial guess for $R(x)$. Second, tabulate the resulting deviations, find the mean absolute deviation, call it r, and then solve (again in the least-squares sense) equation (5.13.5) with r fixed and the $\pm$ chosen to be the sign of the observed deviation at each point x_i. Third, repeat the second step a few times.

You can spot some Remes orthodoxy lurking in our algorithm: The equations we solve are trying to bring the deviations not to zero, but rather to plus-or-minus some consistent value. However, we dispense with keeping track of actual extrema; and we solve only linear equations at each stage. One additional trick is to solve a *weighted* least-squares problem, where the weights are chosen to beat down the largest deviations fastest.

Here is a program implementing these ideas. Notice that the only calls to the function `fn` occur in the initial filling of the table `fs`. You could easily modify the code to do this filling outside of the routine. It is not even necessary that your abscissas `xs` be exactly the ones that we use, though the quality of the fit will deteriorate if you do not have several abscissas between each extremum of the (underlying) minimax solution. Notice that the rational coefficients are output in a format suitable for evaluation by the routine `ratval` in §5.3.

```
      SUBROUTINE ratlsq(fn,a,b,mm,kk,cof,dev)
      INTEGER kk,mm,NPFAC,MAXC,MAXP,MAXIT
      DOUBLE PRECISION a,b,dev,cof(mm+kk+1),fn,PIO2,BIG
      PARAMETER (NPFAC=8,MAXC=20,MAXP=NPFAC*MAXC+1,
     *     MAXIT=5,PIO2=3.141592653589793D0/2.D0,BIG=1.D30)
      EXTERNAL fn
C     USES fn,ratval,dsvbksb,dsvdcmp     DOUBLE PRECISION versions of svdcmp, svbksb.
        Returns in cof(1:mm+kk+1) the coefficients of a rational function approximation to the
        function fn in the interval (a, b). Input quantities mm and kk specify the order of the numer-
        ator and denominator, respectively. The maximum absolute deviation of the approximation
        (insofar as is known) is returned as dev.
      INTEGER i,it,j,ncof,npt
      DOUBLE PRECISION devmax,e,hth,pow,sum,bb(MAXP),coff(MAXC),ee(MAXP),
     *     fs(MAXP),u(MAXP,MAXC),v(MAXC,MAXC),w(MAXC),wt(MAXP),xs(MAXP),
     *     ratval
      ncof=mm+kk+1
      npt=NPFAC*ncof                  Number of points where function is evaluated, i.e., fineness
      dev=BIG                             of the mesh.
      do 11 i=1,npt                   Fill arrays with mesh abscissas and function values.
        if (i.lt.npt/2) then          At each end, use formula that minimizes roundoff sensitivity.
          hth=PIO2*(i-1)/(npt-1.d0)
          xs(i)=a+(b-a)*sin(hth)**2
        else
          hth=PIO2*(npt-i)/(npt-1.d0)
          xs(i)=b-(b-a)*sin(hth)**2
        endif
        fs(i)=fn(xs(i))
        wt(i)=1.d0                    In later iterations we will adjust these weights to combat the
        ee(i)=1.d0                        largest deviations.
      enddo 11
      e=0.d0
      do 17 it=1,MAXIT                Loop over iterations.
        do 14 i=1,npt                 Set up the "design matrix" for the least-squares fit.
          pow=wt(i)
          bb(i)=pow*(fs(i)+sign(e,ee(i)))      Key idea here: Fit to fn(x) + e where
          do 12 j=1,mm+1                           the deviation is positive, to fn(x) − e
            u(i,j)=pow                             where it is negative. Then e is sup-
            pow=pow*xs(i)                          posed to become an approximation
          enddo 12                                 to the equal-ripple deviation.
          pow=-bb(i)
          do 13 j=mm+2,ncof
            pow=pow*xs(i)
            u(i,j)=pow
          enddo 13
        enddo 14
        call dsvdcmp(u,npt,ncof,MAXP,MAXC,w,v)     Singular Value Decomposition.
```

```
        In especially singular or difficult cases, one might here edit the singular values w(1:ncof),
        replacing small values by zero.
      call dsvbksb(u,w,v,npt,ncof,MAXP,MAXC,bb,coff)
      devmax=0.d0
      sum=0.d0
      do 15 j=1,npt                Tabulate the deviations and revise the weights.
        ee(j)=ratval(xs(j),coff,mm,kk)-fs(j)
        wt(j)=abs(ee(j))           Use weighting to emphasize most deviant points.
        sum=sum+wt(j)
        if(wt(j).gt.devmax)devmax=wt(j)
      enddo 15
      e=sum/npt                    Update e to be the mean absolute deviation.
      if (devmax.le.dev) then      Save only the best coefficient set found.
        do 16 j=1,ncof
          cof(j)=coff(j)
        enddo 16
        dev=devmax
      endif
      write (*,10) it,devmax
    enddo 17
    return
10  FORMAT (1x,'ratlsq iteration=',i2,' max error=',1pe10.3)
    END
```

Figure 5.13.1 shows the discrepancies for the first five iterations of `ratlsq` when it is applied to find the $m = k = 4$ rational fit to the function $f(x) = \cos x/(1 + e^x)$ in the interval $(0, \pi)$. One sees that after the first iteration, the results are virtually as good as the minimax solution. The iterations do not converge in the order that the figure suggests: In fact, it is the second iteration that is best (has smallest maximum deviation). The routine `ratlsq` accordingly returns the best of its iterations, not necessarily the last one; there is no advantage in doing more than five iterations.

CITED REFERENCES AND FURTHER READING:

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), Chapter 13. [1]

5.14 Evaluation of Functions by Path Integration

In computer programming, the technique of choice is not necessarily the most efficient, or elegant, or fastest executing one. Instead, it may be the one that is quick to implement, general, and easy to check.

One sometimes needs only a few, or a few thousand, evaluations of a special function, perhaps a complex valued function of a complex variable, that has many different parameters, or asymptotic regimes, or both. Use of the usual tricks (series, continued fractions, rational function approximations, recurrence relations, and so forth) may result in a patchwork program with tests and branches to different formulas. While such a program may be highly efficient in execution, it is often not the shortest way to the answer from a standing start.

A different technique of considerable generality is direct integration of a function's defining differential equation – an ab initio integration for each desired

```
        In especially singular or difficult cases, one might here edit the singular values w(1:ncof),
        replacing small values by zero.
      call dsvbksb(u,w,v,npt,ncof,MAXP,MAXC,bb,coff)
      devmax=0.d0
      sum=0.d0
      do 15 j=1,npt                Tabulate the deviations and revise the weights.
        ee(j)=ratval(xs(j),coff,mm,kk)-fs(j)
        wt(j)=abs(ee(j))           Use weighting to emphasize most deviant points.
        sum=sum+wt(j)
        if(wt(j).gt.devmax)devmax=wt(j)
      enddo 15
      e=sum/npt                    Update e to be the mean absolute deviation.
      if (devmax.le.dev) then      Save only the best coefficient set found.
        do 16 j=1,ncof
          cof(j)=coff(j)
        enddo 16
        dev=devmax
      endif
      write (*,10) it,devmax
    enddo 17
    return
10  FORMAT (1x,'ratlsq iteration=',i2,' max error=',1pe10.3)
    END
```

Figure 5.13.1 shows the discrepancies for the first five iterations of `ratlsq` when it is applied to find the $m = k = 4$ rational fit to the function $f(x) = \cos x/(1 + e^x)$ in the interval $(0, \pi)$. One sees that after the first iteration, the results are virtually as good as the minimax solution. The iterations do not converge in the order that the figure suggests: In fact, it is the second iteration that is best (has smallest maximum deviation). The routine `ratlsq` accordingly returns the best of its iterations, not necessarily the last one; there is no advantage in doing more than five iterations.

CITED REFERENCES AND FURTHER READING:

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), Chapter 13. [1]

5.14 Evaluation of Functions by Path Integration

In computer programming, the technique of choice is not necessarily the most efficient, or elegant, or fastest executing one. Instead, it may be the one that is quick to implement, general, and easy to check.

One sometimes needs only a few, or a few thousand, evaluations of a special function, perhaps a complex valued function of a complex variable, that has many different parameters, or asymptotic regimes, or both. Use of the usual tricks (series, continued fractions, rational function approximations, recurrence relations, and so forth) may result in a patchwork program with tests and branches to different formulas. While such a program may be highly efficient in execution, it is often not the shortest way to the answer from a standing start.

A different technique of considerable generality is direct integration of a function's defining differential equation – an ab initio integration for each desired

function value — along a path in the complex plane if necessary. While this may at first seem like swatting a fly with a golden brick, it turns out that when you already have the brick, and the fly is asleep right under it, all you have to do is let it fall!

As a specific example, let us consider the complex hypergeometric function ${}_2F_1(a,b,c;z)$, which is defined as the analytic continuation of the so-called hypergeometric series,

$$
\begin{aligned}
{}_2F_1(a,b,c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots \\
+ \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots
\end{aligned}
\tag{5.14.1}
$$

The series converges only within the unit circle $|z| < 1$ (see [1]), but one's interest in the function is often not confined to this region.

The hypergeometric function ${}_2F_1$ is a solution (in fact *the* solution that is regular at the origin) of the hypergeometric differential equation, which we can write as

$$
z(1-z)F'' = abF - [c-(a+b+1)z]F' \tag{5.14.2}
$$

Here prime denotes d/dz. One can see that the equation has regular singular points at $z = 0, 1$, and ∞. Since the desired solution is regular at $z = 0$, the values 1 and ∞ will in general be branch points. If we want ${}_2F_1$ to be a single valued function, we must have a branch cut connecting these two points. A conventional position for this cut is along the positive real axis from 1 to ∞, though we may wish to keep open the possibility of altering this choice for some applications.

Our golden brick consists of a collection of routines for the integration of sets of ordinary differential equations, which we will develop in detail later, in Chapter 16. For now, we need only a high-level, "black-box" routine that integrates such a set from initial conditions at one value of a (real) independent variable to final conditions at some other value of the independent variable, while automatically adjusting its internal stepsize to maintain some specified accuracy. That routine is called `odeint` and, in one particular invocation, calculates its individual steps with a sophisticated Bulirsch-Stoer technique.

Suppose that we know values for F and its derivative F' at some value z_0, and that we want to find F at some other point z_1 in the complex plane. The straight-line path connecting these two points is parametrized by

$$
z(s) = z_0 + s(z_1 - z_0) \tag{5.14.3}
$$

with s a real parameter. The differential equation (5.14.2) can now be written as a set of two first-order equations,

$$
\begin{aligned}
\frac{dF}{ds} &= (z_1 - z_0)F' \\
\frac{dF'}{ds} &= (z_1 - z_0)\left(\frac{abF - [c-(a+b+1)z]F'}{z(1-z)}\right)
\end{aligned}
\tag{5.14.4}
$$

to be integrated from $s = 0$ to $s = 1$. Here F and F' are to be viewed as two independent complex variables. The fact that prime means d/dz can be ignored; it

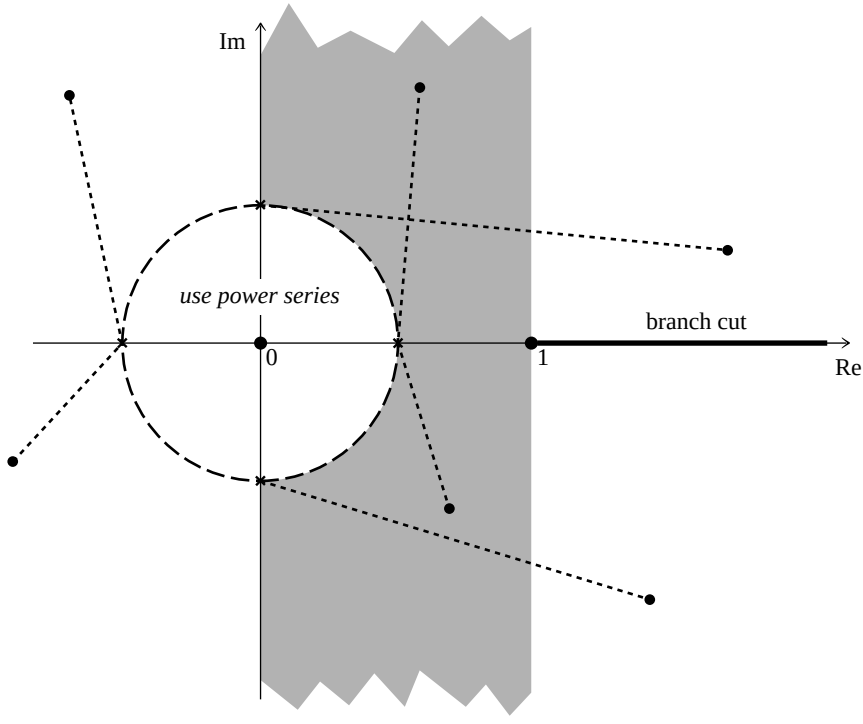


Figure 5.14.1. Complex plane showing the singular points of the hypergeometric function, its branch cut, and some integration paths from the circle $|z| = 1/2$ (where the power series converges rapidly) to other points in the plane.

will emerge as a consequence of the first equation in (5.14.4). Moreover, the real and imaginary parts of equation (5.14.4) define a set of four *real* differential equations, with independent variable s. The complex arithmetic on the right-hand side can be viewed as mere shorthand for how the four components are to be coupled. It is precisely this point of view that gets passed to the routine `odeint`, since it knows nothing of either complex functions or complex independent variables.

It remains only to decide where to start, and what path to take in the complex plane, to get to an arbitrary point z. This is where consideration of the function's singularities, and the adopted branch cut, enter. Figure 5.14.1 shows the strategy that we adopt. For $|z| \leq 1/2$, the series in equation (5.14.1) will in general converge rapidly, and it makes sense to use it directly. Otherwise, we integrate along a straight line path from one of the starting points $(\pm 1/2, 0)$ or $(0, \pm 1/2)$. The former choices are natural for $0 < \mathrm{Re}(z) < 1$ and $\mathrm{Re}(z) < 0$, respectively. The latter choices are used for $\mathrm{Re}(z) > 1$, above and below the branch cut; the purpose of starting away from the real axis in these cases is to avoid passing too close to the singularity at $z = 1$ (see Figure 5.14.1). The location of the branch cut is *defined* by the fact that our adopted strategy never integrates across the real axis for $\mathrm{Re}\,(z) > 1$.

An implementation of this algorithm is given in §6.12 as the routine `hypgeo`.

A number of variants on the procedure described thus far are possible, and easy to program. If successively called values of z are close together (with identical values of a, b, and c), then you can save the state vector (F, F') and the corresponding value

of z on each call, and use these as starting values for the next call. The incremental integration may then take only one or two steps. Avoid integrating across the branch cut unintentionally: the function value will be "correct," but not the one you want.

Alternatively, you may wish to integrate to some position z by a dog-leg path that *does* cross the real axis $\mathrm{Re}\, z > 1$, as a means of *moving* the branch cut. For example, in some cases you might want to integrate from $(0, 1/2)$ to $(3/2, 1/2)$, and go from there to any point with $\mathrm{Re}\, z > 1$ — with either sign of $\mathrm{Im}\, z$. (If you are, for example, finding roots of a function by an iterative method, you do not want the integration for nearby values to take different paths around a branch point. If it does, your root-finder will see discontinuous function values, and will likely not converge correctly!)

In any case, be aware that a loss of numerical accuracy can result if you integrate through a region of large function value on your way to a final answer where the function value is small. (For the hypergeometric function, a particular case of this is when a and b are both large and positive, with c and $x \gtrsim 1$.) In such cases, you'll need to find a better dog-leg path.

The general technique of evaluating a function by integrating its differential equation in the complex plane can also be applied to other special functions. For example, the complex Bessel function, Airy function, Coulomb wave function, and Weber function are all special cases of the *confluent hypergeometric function*, with a differential equation similar to the one used above (see, e.g., [1] §13.6, for a table of special cases). The confluent hypergeometric function has no singularities at finite z: That makes it easy to integrate. However, its essential singularity at infinity means that it can have, along some paths and for some parameters, highly oscillatory or exponentially decreasing behavior: That makes it hard to integrate. Some case by case judgment (or experimentation) is therefore required.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [1]

Chapter 6. Special Functions

6.0 Introduction

There is nothing particularly special about a *special function*, except that some person in authority or textbook writer (not the same thing!) has decided to bestow the moniker. Special functions are sometimes called *higher transcendental functions* (higher than what?) or *functions of mathematical physics* (but they occur in other fields also) or *functions that satisfy certain frequently occurring second-order differential equations* (but not all special functions do). One might simply call them "useful functions" and let it go at that; it is surely only a matter of taste which functions we have chosen to include in this chapter.

Good commercially available program libraries, such as NAG or IMSL, contain routines for a number of special functions. These routines are intended for users who will have no idea what goes on inside them. Such state of the art "black boxes" are often very messy things, full of branches to completely different methods depending on the value of the calling arguments. Black boxes have, or should have, careful control of accuracy, to some stated uniform precision in all regimes.

We will not be quite so fastidious in our examples, in part because we want to illustrate techniques from Chapter 5, and in part because we *want* you to understand what goes on in the routines presented. Some of our routines have an accuracy parameter that can be made as small as desired, while others (especially those involving polynomial fits) give only a certain accuracy, one that we believe serviceable (typically six significant figures or more). We do *not* certify that the routines are perfect black boxes. We do hope that, if you ever encounter trouble in a routine, you will be able to diagnose and correct the problem on the basis of the information that we have given.

In short, the special function routines of this chapter are meant to be used — we use them all the time — but we also want you to be prepared to understand their inner workings.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York) [full of useful numerical approximations to a great variety of functions].

IMSL Sfun/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042).

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.), Chapter S.

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley).
Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press).
Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press).

6.1 Gamma Function, Beta Function, Factorials, Binomial Coefficients

The gamma function is defined by the integral

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \tag{6.1.1}$$

When the argument z is an integer, the gamma function is just the familiar factorial function, but offset by one,

$$n! = \Gamma(n+1) \tag{6.1.2}$$

The gamma function satisfies the recurrence relation

$$\Gamma(z+1) = z\Gamma(z) \tag{6.1.3}$$

If the function is known for arguments $z > 1$ or, more generally, in the half complex plane $\mathrm{Re}(z) > 1$ it can be obtained for $z < 1$ or $\mathrm{Re}\,(z) < 1$ by the reflection formula

$$\Gamma(1-z) = \frac{\pi}{\Gamma(z)\sin(\pi z)} = \frac{\pi z}{\Gamma(1+z)\sin(\pi z)} \tag{6.1.4}$$

Notice that $\Gamma(z)$ has a pole at $z = 0$, and at all negative integer values of z.

There are a variety of methods in use for calculating the function $\Gamma(z)$ numerically, but none is quite as neat as the approximation derived by Lanczos [1]. This scheme is entirely specific to the gamma function, seemingly plucked from thin air. We will not attempt to derive the approximation, but only state the resulting formula: For certain integer choices of γ and N, and for certain coefficients $c_1, c_2, \ldots, c_N$, the gamma function is given by

$$\begin{aligned}\Gamma(z+1) &= (z+\gamma+\tfrac{1}{2})^{z+\frac{1}{2}} e^{-(z+\gamma+\frac{1}{2})} \\ &\quad \times \sqrt{2\pi}\left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon\right] \quad (z > 0)\end{aligned} \tag{6.1.5}$$

You can see that this is a sort of take-off on Stirling's approximation, but with a series of corrections that take into account the first few poles in the left complex plane. The constant c_0 is very nearly equal to 1. The error term is parametrized by ϵ. For $\gamma = 5$, $N = 6$, and a certain set of c's, the error is smaller than $|\epsilon| < 2 \times 10^{-10}$. Impressed? If not, then perhaps you will be impressed by the fact that (with these

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley).
Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press).
Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press).

6.1 Gamma Function, Beta Function, Factorials, Binomial Coefficients

The gamma function is defined by the integral

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \tag{6.1.1}$$

When the argument z is an integer, the gamma function is just the familiar factorial function, but offset by one,

$$n! = \Gamma(n+1) \tag{6.1.2}$$

The gamma function satisfies the recurrence relation

$$\Gamma(z+1) = z\Gamma(z) \tag{6.1.3}$$

If the function is known for arguments $z > 1$ or, more generally, in the half complex plane $\mathrm{Re}(z) > 1$ it can be obtained for $z < 1$ or $\mathrm{Re}\,(z) < 1$ by the reflection formula

$$\Gamma(1-z) = \frac{\pi}{\Gamma(z)\sin(\pi z)} = \frac{\pi z}{\Gamma(1+z)\sin(\pi z)} \tag{6.1.4}$$

Notice that $\Gamma(z)$ has a pole at $z = 0$, and at all negative integer values of z.

There are a variety of methods in use for calculating the function $\Gamma(z)$ numerically, but none is quite as neat as the approximation derived by Lanczos [1]. This scheme is entirely specific to the gamma function, seemingly plucked from thin air. We will not attempt to derive the approximation, but only state the resulting formula: For certain integer choices of γ and N, and for certain coefficients $c_1, c_2, \ldots, c_N$, the gamma function is given by

$$\begin{aligned}\Gamma(z+1) &= (z+\gamma+\tfrac{1}{2})^{z+\frac{1}{2}} e^{-(z+\gamma+\frac{1}{2})} \\ &\quad \times \sqrt{2\pi}\left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon\right] \quad (z > 0)\end{aligned} \tag{6.1.5}$$

You can see that this is a sort of take-off on Stirling's approximation, but with a series of corrections that take into account the first few poles in the left complex plane. The constant c_0 is very nearly equal to 1. The error term is parametrized by ϵ. For $\gamma = 5$, $N = 6$, and a certain set of c's, the error is smaller than $|\epsilon| < 2 \times 10^{-10}$. Impressed? If not, then perhaps you will be impressed by the fact that (with these

same parameters) the formula (6.1.5) and bound on ϵ apply for the *complex* gamma function, *everywhere in the half complex plane Re* $z > 0$.

It is better to implement $\ln\Gamma(x)$ than $\Gamma(x)$, since the latter will overflow many computers' floating-point representation at quite modest values of x. Often the gamma function is used in calculations where the large values of $\Gamma(x)$ are divided by other large numbers, with the result being a perfectly ordinary value. Such operations would normally be coded as subtraction of logarithms. With (6.1.5) in hand, we can compute the logarithm of the gamma function with two calls to a logarithm and 25 or so arithmetic operations. This makes it not much more difficult than other built-in functions that we take for granted, such as $\sin x$ or e^x:

```
      FUNCTION gammln(xx)
      REAL gammln,xx
        Returns the value ln[Γ(xx)] for xx > 0.
      INTEGER j
      DOUBLE PRECISION ser,stp,tmp,x,y,cof(6)
        Internal arithmetic will be done in double precision, a nicety that you can omit if five-figure
        accuracy is good enough.
      SAVE cof,stp
      DATA cof,stp/76.18009172947146d0,-86.50532032941677d0,
     *  24.01409824083091d0,-1.231739572450155d0,.1208650973866179d-2,
     *  -.5395239384953d-5,2.5066282746310005d0/
      x=xx
      y=x
      tmp=x+5.5d0
      tmp=(x+0.5d0)*log(tmp)-tmp
      ser=1.000000000190015d0
      do 11 j=1,6
        y=y+1.d0
        ser=ser+cof(j)/y
      enddo 11
      gammln=tmp+log(stp*ser/x)
      return
      END
```

How shall we write a routine for the factorial function $n!$? Generally the factorial function will be called for small integer values (for large values it will overflow anyway!), and in most applications the same integer value will be called for many times. It is a profligate waste of computer time to call `exp(gammln(n+1.0))` for each required factorial. Better to go back to basics, holding `gammln` in reserve for unlikely calls:

```
      FUNCTION factrl(n)
      INTEGER n
      REAL factrl
C     USES gammln
        Returns the value n! as a floating-point number.
      INTEGER j,ntop
      REAL a(33),gammln            Table to be filled in only as required.
      SAVE ntop,a
      DATA ntop,a(1)/0,1./         Table initialized with 0! only.
      if (n.lt.0) then
        pause 'negative factorial in factrl'
      else if (n.le.ntop) then     Already in table.
        factrl=a(n+1)
      else if (n.le.32) then       Fill in table up to desired value.
        do 11 j=ntop+1,n
```

```
            a(j+1)=j*a(j)
        enddo 11
        ntop=n
        factrl=a(n+1)
    else                                Larger value than size of table is required. Actually, this big
        factrl=exp(gammln(n+1.))            a value is going to overflow on many computers, but no
    endif                                   harm in trying.
    return
    END
```

A useful point is that `factrl` will be *exact* for the smaller values of `n`, since floating-point multiplies on small integers are exact on all computers. This exactness will not hold if we turn to the logarithm of the factorials. For binomial coefficients, however, we must do exactly this, since the individual factorials in a binomial coefficient will overflow long before the coefficient itself will.

The binomial coefficient is defined by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad 0 \le k \le n \tag{6.1.6}$$

```
    FUNCTION bico(n,k)
    INTEGER k,n
    REAL bico
C   USES factln
        Returns the binomial coefficient (n k) as a floating-point number.
    REAL factln
    bico=nint(exp(factln(n)-factln(k)-factln(n-k)))
    return  The nearest-integer function cleans up roundoff error for smaller values of n and k.
    END
```

which uses

```
    FUNCTION factln(n)
    INTEGER n
    REAL factln
C   USES gammln
        Returns ln(n!).
    REAL a(100),gammln
    SAVE a
    DATA a/100*-1./                             Initialize the table to negative values.
    if (n.lt.0) pause 'negative factorial in factln'
    if (n.le.99) then                           In range of the table.
        if (a(n+1).lt.0.) a(n+1)=gammln(n+1.)       If not already in the table, put it in.
        factln=a(n+1)
    else
        factln=gammln(n+1.)                     Out of range of the table.
    endif
    return
    END
```

If your problem requires a series of related binomial coefficients, a good idea is to use recurrence relations, for example

$$\begin{aligned}\binom{n+1}{k} &= \frac{n+1}{n-k+1}\binom{n}{k} = \binom{n}{k} + \binom{n}{k-1} \\ \binom{n}{k+1} &= \frac{n-k}{k+1}\binom{n}{k}\end{aligned} \tag{6.1.7}$$

Finally, turning away from the combinatorial functions with integer valued arguments, we come to the beta function,

$$B(z,w) = B(w,z) = \int_0^1 t^{z-1}(1-t)^{w-1}dt \tag{6.1.8}$$

which is related to the gamma function by

$$B(z,w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \tag{6.1.9}$$

hence

```
      FUNCTION beta(z,w)
      REAL beta,w,z
C     USES gammln
        Returns the value of the beta function B(z,w).
      REAL gammln
      beta=exp(gammln(z)+gammln(w)-gammln(z+w))
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 6.

Lanczos, C. 1964, *SIAM Journal on Numerical Analysis*, ser. B, vol. 1, pp. 86–96. [1]

6.2 Incomplete Gamma Function, Error Function, Chi-Square Probability Function, Cumulative Poisson Function

The incomplete gamma function is defined by

$$P(a,x) \equiv \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}dt \qquad (a>0) \tag{6.2.1}$$

If your problem requires a series of related binomial coefficients, a good idea is to use recurrence relations, for example

$$\begin{aligned}\binom{n+1}{k} &= \frac{n+1}{n-k+1}\binom{n}{k} = \binom{n}{k} + \binom{n}{k-1} \\ \binom{n}{k+1} &= \frac{n-k}{k+1}\binom{n}{k}\end{aligned} \tag{6.1.7}$$

Finally, turning away from the combinatorial functions with integer valued arguments, we come to the beta function,

$$B(z,w) = B(w,z) = \int_0^1 t^{z-1}(1-t)^{w-1}dt \tag{6.1.8}$$

which is related to the gamma function by

$$B(z,w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \tag{6.1.9}$$

hence

```
      FUNCTION beta(z,w)
      REAL beta,w,z
C     USES gammln
        Returns the value of the beta function B(z,w).
      REAL gammln
      beta=exp(gammln(z)+gammln(w)-gammln(z+w))
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 6.

Lanczos, C. 1964, *SIAM Journal on Numerical Analysis*, ser. B, vol. 1, pp. 86–96. [1]

6.2 Incomplete Gamma Function, Error Function, Chi-Square Probability Function, Cumulative Poisson Function

The incomplete gamma function is defined by

$$P(a,x) \equiv \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}dt \qquad (a>0) \tag{6.2.1}$$

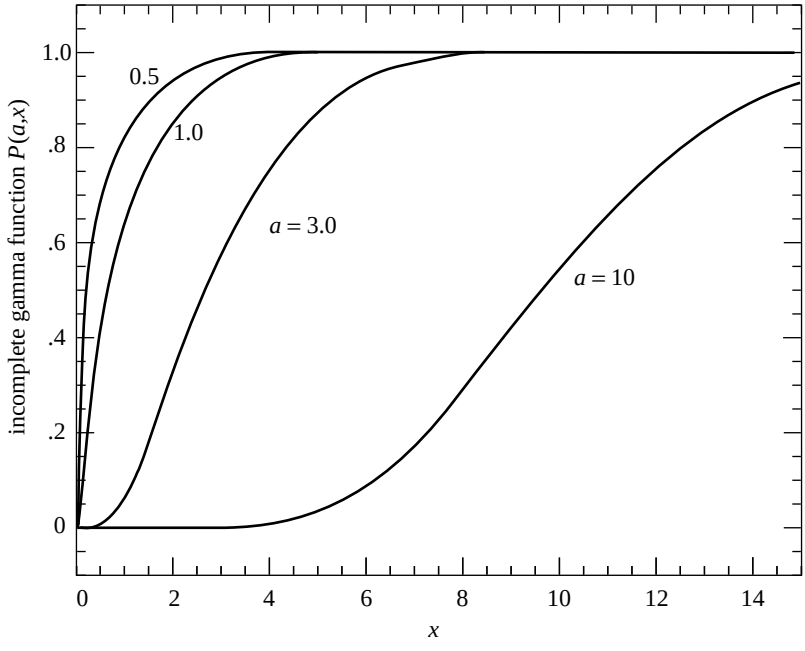


Figure 6.2.1. The incomplete gamma function $P(a,x)$ for four values of a.

It has the limiting values

$$P(a,0)=0 \qquad \text{and} \qquad P(a,\infty)=1 \tag{6.2.2}$$

The incomplete gamma function $P(a,x)$ is monotonic and (for a greater than one or so) rises from "near-zero" to "near-unity" in a range of x centered on about $a-1$, and of width about $\sqrt{a}$ (see Figure 6.2.1).

The complement of $P(a,x)$ is also confusingly called an incomplete gamma function,

$$Q(a,x) \equiv 1-P(a,x) \equiv \frac{\Gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)}\int_x^\infty e^{-t}t^{a-1}dt \qquad (a>0) \tag{6.2.3}$$

It has the limiting values

$$Q(a,0)=1 \qquad \text{and} \qquad Q(a,\infty)=0 \tag{6.2.4}$$

The notations $P(a,x)$, $\gamma(a,x)$, and $\Gamma(a,x)$ are standard; the notation $Q(a,x)$ is specific to this book.

There is a series development for $\gamma(a,x)$ as follows:

$$\gamma(a,x) = e^{-x}x^a \sum_{n=0}^{\infty} \frac{\Gamma(a)}{\Gamma(a+1+n)}x^n \tag{6.2.5}$$

One does not actually need to compute a new $\Gamma(a+1+n)$ for each n; one rather uses equation (6.1.3) and the previous coefficient.

A continued fraction development for $\Gamma(a,x)$ is

$$\Gamma(a,x) = e^{-x}x^a\left(\frac{1}{x+}\ \frac{1-a}{1+}\ \frac{1}{x+}\ \frac{2-a}{1+}\ \frac{2}{x+}\cdots\right) \qquad (x>0) \qquad (6.2.6)$$

It is computationally better to use the even part of (6.2.6), which converges twice as fast (see §5.2):

$$\Gamma(a,x) = e^{-x}x^a\left(\frac{1}{x+1-a-}\ \frac{1\cdot(1-a)}{x+3-a-}\ \frac{2\cdot(2-a)}{x+5-a-}\ \cdots\right) \qquad (x>0) \qquad (6.2.7)$$

It turns out that (6.2.5) converges rapidly for x less than about $a+1$, while (6.2.6) or (6.2.7) converges rapidly for x greater than about $a+1$. In these respective regimes each requires at most a few times $\sqrt{a}$ terms to converge, and this many only near $x=a$, where the incomplete gamma functions are varying most rapidly. Thus (6.2.5) and (6.2.7) together allow evaluation of the function for all positive a and x. An extra dividend is that we never need compute a function value near zero by subtracting two nearly equal numbers. The higher-level functions that return $P(a,x)$ and $Q(a,x)$ are

```
      FUNCTION gammp(a,x)
      REAL a,gammp,x
C     USES gcf,gser
        Returns the incomplete gamma function P(a,x).
      REAL gammcf,gamser,gln
      if(x.lt.0..or.a.le.0.)pause 'bad arguments in gammp'
      if(x.lt.a+1.)then           Use the series representation.
        call gser(gamser,a,x,gln)
        gammp=gamser
      else                        Use the continued fraction representation
        call gcf(gammcf,a,x,gln)
        gammp=1.-gammcf           and take its complement.
      endif
      return
      END
```

```
      FUNCTION gammq(a,x)
      REAL a,gammq,x
C     USES gcf,gser
        Returns the incomplete gamma function Q(a,x) ≡ 1 − P(a,x).
      REAL gammcf,gamser,gln
      if(x.lt.0..or.a.le.0.)pause 'bad arguments in gammq'
      if(x.lt.a+1.)then           Use the series representation
        call gser(gamser,a,x,gln)
        gammq=1.-gamser           and take its complement.
      else                        Use the continued fraction representation.
        call gcf(gammcf,a,x,gln)
        gammq=gammcf
      endif
      return
      END
```

The argument gln is returned by both the series and continued fraction procedures containing the value $\ln\Gamma(a)$; the reason for this is so that it is available to you if you want to modify the above two procedures to give $\gamma(a,x)$ and $\Gamma(a,x)$, in addition to $P(a,x)$ and $Q(a,x)$ (cf. equations 6.2.1 and 6.2.3).

The procedures gser and gcf which implement (6.2.5) and (6.2.7) are

```
      SUBROUTINE gser(gamser,a,x,gln)
      INTEGER ITMAX
      REAL a,gamser,gln,x,EPS
      PARAMETER (ITMAX=100,EPS=3.e-7)
C     USES gammln
        Returns the incomplete gamma function P(a,x) evaluated by its series representation as
        gamser. Also returns ln Γ(a) as gln.
      INTEGER n
      REAL ap,del,sum,gammln
      gln=gammln(a)
      if(x.le.0.)then
          if(x.lt.0.)pause 'x < 0 in gser'
          gamser=0.
          return
      endif
      ap=a
      sum=1./a
      del=sum
      do 11 n=1,ITMAX
          ap=ap+1.
          del=del*x/ap
          sum=sum+del
          if(abs(del).lt.abs(sum)*EPS)goto 1
      enddo 11
      pause 'a too large, ITMAX too small in gser'
1     gamser=sum*exp(-x+a*log(x)-gln)
      return
      END
```

```
      SUBROUTINE gcf(gammcf,a,x,gln)
      INTEGER ITMAX
      REAL a,gammcf,gln,x,EPS,FPMIN
      PARAMETER (ITMAX=100,EPS=3.e-7,FPMIN=1.e-30)
C     USES gammln
        Returns the incomplete gamma function Q(a,x) evaluated by its continued fraction repre-
        sentation as gammcf. Also returns ln Γ(a) as gln.
        Parameters: ITMAX is the maximum allowed number of iterations; EPS is the relative accu-
        racy; FPMIN is a number near the smallest representable floating-point number.
      INTEGER i
      REAL an,b,c,d,del,h,gammln
      gln=gammln(a)
      b=x+1.-a                          Set up for evaluating continued fraction by modified
      c=1./FPMIN                            Lentz's method (§5.2) with b_0 = 0.
      d=1./b
      h=d
      do 11 i=1,ITMAX                   Iterate to convergence.
          an=-i*(i-a)
          b=b+2.
          d=an*d+b
          if(abs(d).lt.FPMIN)d=FPMIN
          c=b+an/c
          if(abs(c).lt.FPMIN)c=FPMIN
          d=1./d
          del=d*c
```

```
        h=h*del
        if(abs(del-1.).lt.EPS)goto 1
    enddo 11
    pause 'a too large, ITMAX too small in gcf'
1   gammcf=exp(-x+a*log(x)-gln)*h       Put factors in front.
    return
    END
```

Error Function

The error function and complementary error function are special cases of the incomplete gamma function, and are obtained moderately efficiently by the above procedures. Their definitions are

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{6.2.8}$$

and

$$\mathrm{erfc}(x) \equiv 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt \tag{6.2.9}$$

The functions have the following limiting values and symmetries:

$$\mathrm{erf}(0) = 0 \qquad \mathrm{erf}(\infty) = 1 \qquad \mathrm{erf}(-x) = -\mathrm{erf}(x) \tag{6.2.10}$$

$$\mathrm{erfc}(0) = 1 \qquad \mathrm{erfc}(\infty) = 0 \qquad \mathrm{erfc}(-x) = 2 - \mathrm{erfc}(x) \tag{6.2.11}$$

They are related to the incomplete gamma functions by

$$\mathrm{erf}(x) = P\left(\frac{1}{2}, x^2\right) \qquad (x \geq 0) \tag{6.2.12}$$

and

$$\mathrm{erfc}(x) = Q\left(\frac{1}{2}, x^2\right) \qquad (x \geq 0) \tag{6.2.13}$$

Hence we have

```
    FUNCTION erf(x)
    REAL erf,x
C   USES gammp
        Returns the error function erf(x).
    REAL gammp
    if(x.lt.0.)then
        erf=-gammp(.5,x**2)
    else
        erf=gammp(.5,x**2)
    endif
    return
    END
```

```
      FUNCTION erfc(x)
      REAL erfc,x
C     USES gammp,gammq
        Returns the complementary error function erfc(x).
      REAL gammp,gammq
      if(x.lt.0.)then
          erfc=1.+gammp(.5,x**2)
      else
          erfc=gammq(.5,x**2)
      endif
      return
      END
```

If you care to do so, you can easily remedy the minor inefficiency in `erf` and `erfc`, namely that $\Gamma(0.5) = \sqrt{\pi}$ is computed unnecessarily when `gammp` or `gammq` is called. Before you do that, however, you might wish to consider the following routine, based on Chebyshev fitting to an inspired guess as to the functional form:

```
      FUNCTION erfcc(x)
      REAL erfcc,x
        Returns the complementary error function erfc(x) with fractional error everywhere less than
        1.2 × 10^−7.
      REAL t,z
      z=abs(x)
      t=1./(1.+0.5*z)
      erfcc=t*exp(-z*z-1.26551223+t*(1.00002368+t*(.37409196+
     *     t*(.09678418+t*(-.18628806+t*(.27886807+t*(-1.13520398+
     *     t*(1.48851587+t*(-.82215223+t*.17087277)))))))))
      if (x.lt.0.) erfcc=2.-erfcc
      return
      END
```

There are also some functions of *two* variables that are special cases of the incomplete gamma function:

Cumulative Poisson Probability Function

$P_x(< k)$, for positive x and integer $k \geq 1$, denotes the *cumulative Poisson probability* function. It is defined as the probability that the number of Poisson random events occurring will be between 0 and $k-1$ *inclusive*, if the expected mean number is x. It has the limiting values

$$P_x(< 1) = e^{-x} \qquad P_x(< \infty) = 1 \tag{6.2.14}$$

Its relation to the incomplete gamma function is simply

$$P_x(< k) = Q(k, x) = \texttt{gammq}\,(k, x) \tag{6.2.15}$$

Chi-Square Probability Function

$P(\chi^2|\nu)$ is defined as the probability that the observed chi-square for a correct model should be less than a value χ^2. (We will discuss the use of this function in Chapter 15.) Its complement $Q(\chi^2|\nu)$ is the probability that the observed chi-square will exceed the value χ^2 by chance *even* for a correct model. In both cases ν is an integer, the number of degrees of freedom. The functions have the limiting values

$$P(0|\nu) = 0 \qquad P(\infty|\nu) = 1 \tag{6.2.16}$$

$$Q(0|\nu) = 1 \qquad Q(\infty|\nu) = 0 \tag{6.2.17}$$

and the following relation to the incomplete gamma functions,

$$P(\chi^2|\nu) = P\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) = \texttt{gammp}\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) \tag{6.2.18}$$

$$Q(\chi^2|\nu) = Q\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) = \texttt{gammq}\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) \tag{6.2.19}$$

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 6, 7, and 26.

Pearson, K. (ed.) 1951, *Tables of the Incomplete Gamma Function* (Cambridge: Cambridge University Press).

6.3 Exponential Integrals

The standard definition of the exponential integral is

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt, \qquad x > 0, \quad n = 0, 1, \ldots \tag{6.3.1}$$

The function defined by the principal value of the integral

$$\mathrm{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\, dt = \int_{-\infty}^{x} \frac{e^t}{t}\, dt, \qquad x > 0 \tag{6.3.2}$$

is also called an exponential integral. Note that $\mathrm{Ei}(-x)$ is related to $-E_1(x)$ by analytic continuation.

The function $E_n(x)$ is a special case of the incomplete gamma function

$$E_n(x) = x^{n-1}\Gamma(1-n, x) \tag{6.3.3}$$

Chi-Square Probability Function

$P(\chi^2|\nu)$ is defined as the probability that the observed chi-square for a correct model should be less than a value χ^2. (We will discuss the use of this function in Chapter 15.) Its complement $Q(\chi^2|\nu)$ is the probability that the observed chi-square will exceed the value χ^2 by chance *even* for a correct model. In both cases ν is an integer, the number of degrees of freedom. The functions have the limiting values

$$P(0|\nu) = 0 \qquad P(\infty|\nu) = 1 \tag{6.2.16}$$

$$Q(0|\nu) = 1 \qquad Q(\infty|\nu) = 0 \tag{6.2.17}$$

and the following relation to the incomplete gamma functions,

$$P(\chi^2|\nu) = P\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) = \texttt{gammp}\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) \tag{6.2.18}$$

$$Q(\chi^2|\nu) = Q\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) = \texttt{gammq}\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right) \tag{6.2.19}$$

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 6, 7, and 26.

Pearson, K. (ed.) 1951, *Tables of the Incomplete Gamma Function* (Cambridge: Cambridge University Press).

6.3 Exponential Integrals

The standard definition of the exponential integral is

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt, \qquad x > 0, \quad n = 0, 1, \ldots \tag{6.3.1}$$

The function defined by the principal value of the integral

$$\mathrm{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\, dt = \int_{-\infty}^{x} \frac{e^t}{t}\, dt, \qquad x > 0 \tag{6.3.2}$$

is also called an exponential integral. Note that $\mathrm{Ei}(-x)$ is related to $-E_1(x)$ by analytic continuation.

The function $E_n(x)$ is a special case of the incomplete gamma function

$$E_n(x) = x^{n-1}\Gamma(1-n, x) \tag{6.3.3}$$

We can therefore use a similar strategy for evaluating it. The continued fraction — just equation (6.2.6) rewritten — converges for all $x > 0$:

$$E_n(x) = e^{-x}\left(\frac{1}{x+}\,\frac{n}{1+}\,\frac{1}{x+}\,\frac{n+1}{1+}\,\frac{2}{x+}\cdots\right) \tag{6.3.4}$$

We use it in its more rapidly converging even form,

$$E_n(x) = e^{-x}\left(\frac{1}{x+n-}\,\frac{1\cdot n}{x+n+2-}\,\frac{2(n+1)}{x+n+4-}\,\cdots\right) \tag{6.3.5}$$

The continued fraction only really converges fast enough to be useful for $x \gtrsim 1$. For $0 < x \lesssim 1$, we can use the series representation

$$E_n(x) = \frac{(-x)^{n-1}}{(n-1)!}[-\ln x + \psi(n)] - \sum_{\substack{m=0\\ m\neq n-1}}^{\infty} \frac{(-x)^m}{(m-n+1)m!} \tag{6.3.6}$$

The quantity $\psi(n)$ here is the digamma function, given for integer arguments by

$$\psi(1) = -\gamma, \qquad \psi(n) = -\gamma + \sum_{m=1}^{n-1}\frac{1}{m} \tag{6.3.7}$$

where $\gamma = 0.5772156649\ldots$ is Euler's constant. We evaluate the expression (6.3.6) in order of ascending powers of x:

$$\begin{aligned} E_n(x) = &-\left[\frac{1}{(1-n)} - \frac{x}{(2-n)\cdot 1} + \frac{x^2}{(3-n)(1\cdot 2)} - \cdots + \frac{(-x)^{n-2}}{(-1)(n-2)!}\right] \\ &+ \frac{(-x)^{n-1}}{(n-1)!}[-\ln x + \psi(n)] - \left[\frac{(-x)^n}{1\cdot n!} + \frac{(-x)^{n+1}}{2\cdot(n+1)!} + \cdots\right] \end{aligned} \tag{6.3.8}$$

The first square bracket is omitted when $n = 1$. This method of evaluation has the advantage that for large n the series converges before reaching the term containing $\psi(n)$. Accordingly, one needs an algorithm for evaluating $\psi(n)$ only for small n, $n \lesssim 20-40$. We use equation (6.3.7), although a table look-up would improve efficiency slightly.

Amos [1] presents a careful discussion of the truncation error in evaluating equation (6.3.8), and gives a fairly elaborate termination criterion. We have found that simply stopping when the last term added is smaller than the required tolerance works about as well.

Two special cases have to be handled separately:

$$\begin{aligned} E_0(x) &= \frac{e^{-x}}{x} \\ E_n(0) &= \frac{1}{n-1}, \qquad n > 1 \end{aligned} \tag{6.3.9}$$

The routine expint allows fast evaluation of $E_n(x)$ to any accuracy EPS within the reach of your machine's word length for floating-point numbers. The only modification required for increased accuracy is to supply Euler's constant with enough significant digits. Wrench [2] can provide you with the first 328 digits if necessary!

```
FUNCTION expint(n,x)
INTEGER n,MAXIT
REAL expint,x,EPS,FPMIN,EULER
PARAMETER (MAXIT=100,EPS=1.e-7,FPMIN=1.e-30,EULER=.5772156649)
    Evaluates the exponential integral E_n(x).
    Parameters: MAXIT is the maximum allowed number of iterations; EPS is the desired rel-
    ative error, not smaller than the machine precision; FPMIN is a number near the smallest
    representable floating-point number; EULER is Euler's constant γ.
INTEGER i,ii,nm1
REAL a,b,c,d,del,fact,h,psi
nm1=n-1
if(n.lt.0.or.x.lt.0..or.(x.eq.0..and.(n.eq.0.or.n.eq.1)))then
    pause 'bad arguments in expint'
else if(n.eq.0)then                 Special case.
    expint=exp(-x)/x
else if(x.eq.0.)then                Another special case.
    expint=1./nm1
else if(x.gt.1.)then                Lentz's algorithm (§5.2).
    b=x+n
    c=1./FPMIN
    d=1./b
    h=d
    do 11 i=1,MAXIT
        a=-i*(nm1+i)
        b=b+2.
        d=1./(a*d+b)                Denominators cannot be zero.
        c=b+a/c
        del=c*d
        h=h*del
        if(abs(del-1.).lt.EPS)then
            expint=h*exp(-x)
            return
        endif
    enddo 11
    pause 'continued fraction failed in expint'
else                                Evaluate series.
    if(nm1.ne.0)then                Set first term.
        expint=1./nm1
    else
        expint=-log(x)-EULER
    endif
    fact=1.
    do 13 i=1,MAXIT
        fact=-fact*x/i
        if(i.ne.nm1)then
            del=-fact/(i-nm1)
        else
            psi=-EULER              Compute ψ(n).
            do 12 ii=1,nm1
                psi=psi+1./ii
            enddo 12
            del=fact*(-log(x)+psi)
        endif
        expint=expint+del
        if(abs(del).lt.abs(expint)*EPS) return
    enddo 13
```

```
      pause 'series failed in expint'
    endif
    return
    END
```

A good algorithm for evaluating Ei is to use the power series for small x and the asymptotic series for large x. The power series is

$$\mathrm{Ei}(x) = \gamma + \ln x + \frac{x}{1 \cdot 1!} + \frac{x^2}{2 \cdot 2!} + \cdots \tag{6.3.10}$$

where γ is Euler's constant. The asymptotic expansion is

$$\mathrm{Ei}(x) \sim \frac{e^x}{x}\left(1 + \frac{1!}{x} + \frac{2!}{x^2} + \cdots\right) \tag{6.3.11}$$

The lower limit for the use of the asymptotic expansion is approximately $|\ln \mathtt{EPS}|$, where EPS is the required relative error.

```
    FUNCTION ei(x)
    INTEGER MAXIT
    REAL ei,x,EPS,EULER,FPMIN
    PARAMETER (EPS=6.e-8,EULER=.57721566,MAXIT=100,FPMIN=1.e-30)
      Computes the exponential integral Ei(x) for x > 0.
      Parameters: EPS is the relative error, or absolute error near the zero of Ei at x = 0.3725;
      EULER is Euler's constant γ; MAXIT is the maximum number of iterations allowed; FPMIN
      is a number near the smallest representable floating-point number.
    INTEGER k
    REAL fact,prev,sum,term
    if(x.le.0.) pause 'bad argument in ei'
    if(x.lt.FPMIN)then                  Special case: avoid failure of convergence test be-
        ei=log(x)+EULER                     cause of underflow.
    else if(x.le.-log(EPS))then         Use power series.
        sum=0.
        fact=1.
        do 11 k=1,MAXIT
            fact=fact*x/k
            term=fact/k
            sum=sum+term
            if(term.lt.EPS*sum)goto 1
        enddo 11
        pause 'series failed in ei'
1       ei=sum+log(x)+EULER
    else                                Use asymptotic series.
        sum=0.                          Start with second term.
        term=1.
        do 12 k=1,MAXIT
            prev=term
            term=term*k/x
            if(term.lt.EPS)goto 2       Since final sum is greater than one, term itself ap-
            if(term.lt.prev)then            proximates the relative error.
                sum=sum+term            Still converging: add new term.
            else
                sum=sum-prev            Diverging: subtract previous term and exit.
                goto 2
            endif
        enddo 12
2       ei=exp(x)*(1.+sum)/x
    endif
```

```
return
END
```

CITED REFERENCES AND FURTHER READING:

Stegun, I.A., and Zucker, R. 1974, *Journal of Research of the National Bureau of Standards*, vol. 78B, pp. 199–216; 1976, *op. cit.*, vol. 80B, pp. 291–311.

Amos D.E. 1980, *ACM Transactions on Mathematical Software*, vol. 6, pp. 365–377 [1]; also vol. 6, pp. 420–428.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 5.

Wrench J.W. 1952, *Mathematical Tables and Other Aids to Computation*, vol. 6, p. 255. [2]

6.4 Incomplete Beta Function, Student's Distribution, F-Distribution, Cumulative Binomial Distribution

The incomplete beta function is defined by

$$I_x(a,b) \equiv \frac{B_x(a,b)}{B(a,b)} \equiv \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}dt \qquad (a,b>0) \tag{6.4.1}$$

It has the limiting values

$$I_0(a,b) = 0 \qquad I_1(a,b) = 1 \tag{6.4.2}$$

and the symmetry relation

$$I_x(a,b) = 1 - I_{1-x}(b,a) \tag{6.4.3}$$

If a and b are both rather greater than one, then $I_x(a,b)$ rises from "near-zero" to "near-unity" quite sharply at about $x = a/(a+b)$. Figure 6.4.1 plots the function for several pairs (a,b).

The incomplete beta function has a series expansion

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)} \left[1 + \sum_{n=0}^{\infty} \frac{B(a+1,n+1)}{B(a+b,n+1)} x^{n+1} \right], \tag{6.4.4}$$

but this does not prove to be very useful in its numerical evaluation. (Note, however, that the beta functions in the coefficients can be evaluated for each value of n with just the previous value and a few multiplies, using equations 6.1.9 and 6.1.3.)

The continued fraction representation proves to be much more useful,

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)} \left[\frac{1}{1+} \frac{d_1}{1+} \frac{d_2}{1+} \cdots \right] \tag{6.4.5}$$

```
return
END
```

CITED REFERENCES AND FURTHER READING:

Stegun, I.A., and Zucker, R. 1974, *Journal of Research of the National Bureau of Standards*, vol. 78B, pp. 199–216; 1976, *op. cit.*, vol. 80B, pp. 291–311.

Amos D.E. 1980, *ACM Transactions on Mathematical Software*, vol. 6, pp. 365–377 [1]; also vol. 6, pp. 420–428.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 5.

Wrench J.W. 1952, *Mathematical Tables and Other Aids to Computation*, vol. 6, p. 255. [2]

6.4 Incomplete Beta Function, Student's Distribution, F-Distribution, Cumulative Binomial Distribution

The incomplete beta function is defined by

$$I_x(a,b) \equiv \frac{B_x(a,b)}{B(a,b)} \equiv \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}dt \qquad (a,b>0) \tag{6.4.1}$$

It has the limiting values

$$I_0(a,b) = 0 \qquad I_1(a,b) = 1 \tag{6.4.2}$$

and the symmetry relation

$$I_x(a,b) = 1 - I_{1-x}(b,a) \tag{6.4.3}$$

If a and b are both rather greater than one, then $I_x(a,b)$ rises from "near-zero" to "near-unity" quite sharply at about $x = a/(a+b)$. Figure 6.4.1 plots the function for several pairs (a,b).

The incomplete beta function has a series expansion

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)} \left[1 + \sum_{n=0}^{\infty} \frac{B(a+1,n+1)}{B(a+b,n+1)} x^{n+1} \right], \tag{6.4.4}$$

but this does not prove to be very useful in its numerical evaluation. (Note, however, that the beta functions in the coefficients can be evaluated for each value of n with just the previous value and a few multiplies, using equations 6.1.9 and 6.1.3.)

The continued fraction representation proves to be much more useful,

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)} \left[\frac{1}{1+} \frac{d_1}{1+} \frac{d_2}{1+} \cdots \right] \tag{6.4.5}$$

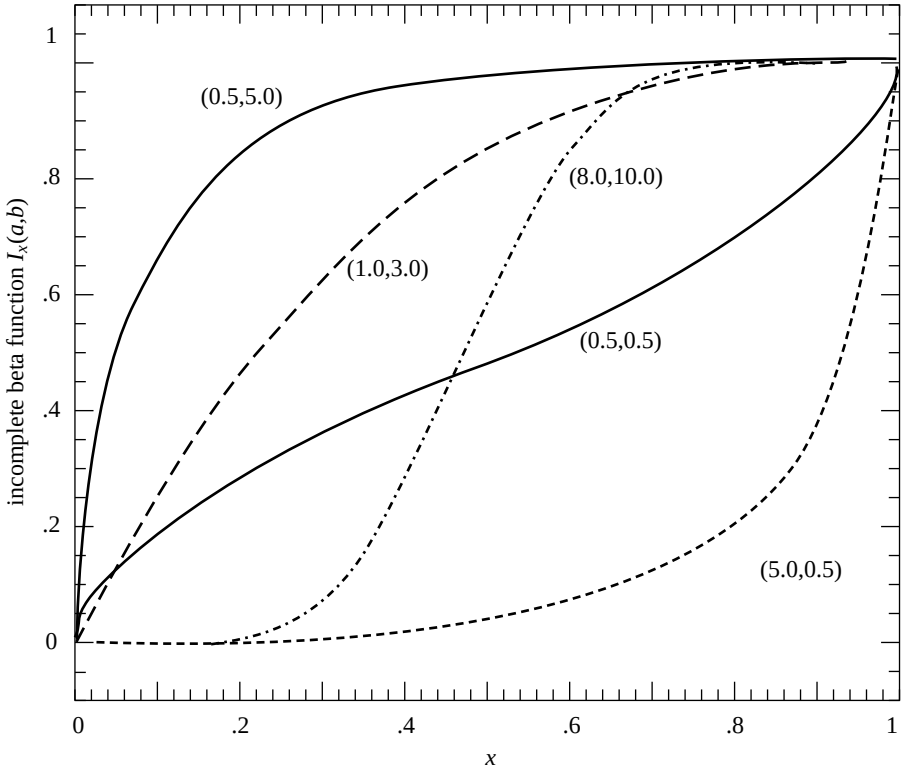


Figure 6.4.1. The incomplete beta function $I_x(a,b)$ for five different pairs of (a,b). Notice that the pairs $(0.5,5.0)$ and $(5.0,0.5)$ are related by reflection symmetry around the diagonal (cf. equation 6.4.3).

where

$$d_{2m+1} = -\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)}$$
$$d_{2m} = \frac{m(b-m)x}{(a+2m-1)(a+2m)} \tag{6.4.6}$$

This continued fraction converges rapidly for $x < (a+1)/(a+b+2)$, taking in the worst case $O(\sqrt{\max(a,b)})$ iterations. But for $x > (a+1)/(a+b+2)$ we can just use the symmetry relation (6.4.3) to obtain an equivalent computation where the continued fraction will also converge rapidly. Hence we have

```
      FUNCTION betai(a,b,x)
      REAL betai,a,b,x
C     USES betacf,gammln
        Returns the incomplete beta function Ix(a,b).
      REAL bt,betacf,gammln
      if(x.lt.0..or.x.gt.1.)pause 'bad argument x in betai'
      if(x.eq.0..or.x.eq.1.)then
        bt=0.
      else                                  Factors in front of the continued fraction.
        bt=exp(gammln(a+b)-gammln(a)-gammln(b)
     *      +a*log(x)+b*log(1.-x))
      endif
      if(x.lt.(a+1.)/(a+b+2.))then          Use continued fraction directly.
```

```
        betai=bt*betacf(a,b,x)/a
        return
    else
        betai=1.-bt*betacf(b,a,1.-x)/b      Use continued fraction after making the symme-
        return                                  try transformation.
    endif
    END
```

which utilizes the continued fraction evaluation routine

```
    FUNCTION betacf(a,b,x)
    INTEGER MAXIT
    REAL betacf,a,b,x,EPS,FPMIN
    PARAMETER (MAXIT=100,EPS=3.e-7,FPMIN=1.e-30)
       Used by betai: Evaluates continued fraction for incomplete beta function by modified
       Lentz's method (§5.2).
    INTEGER m,m2
    REAL aa,c,d,del,h,qab,qam,qap
    qab=a+b                                 These q's will be used in factors that occur in the
    qap=a+1.                                    coefficients (6.4.6).
    qam=a-1.
    c=1.                                    First step of Lentz's method.
    d=1.-qab*x/qap
    if(abs(d).lt.FPMIN)d=FPMIN
    d=1./d
    h=d
    do 11 m=1,MAXIT
        m2=2*m
        aa=m*(b-m)*x/((qam+m2)*(a+m2))
        d=1.+aa*d                           One step (the even one) of the recurrence.
        if(abs(d).lt.FPMIN)d=FPMIN
        c=1.+aa/c
        if(abs(c).lt.FPMIN)c=FPMIN
        d=1./d
        h=h*d*c
        aa=-(a+m)*(qab+m)*x/((a+m2)*(qap+m2))
        d=1.+aa*d                           Next step of the recurrence (the odd one).
        if(abs(d).lt.FPMIN)d=FPMIN
        c=1.+aa/c
        if(abs(c).lt.FPMIN)c=FPMIN
        d=1./d
        del=d*c
        h=h*del
        if(abs(del-1.).lt.EPS)goto 1        Are we done?
    enddo 11
    pause 'a or b too big, or MAXIT too small in betacf'
1   betacf=h
    return
    END
```

Student's Distribution Probability Function

Student's distribution, denoted $A(t|\nu)$, is useful in several statistical contexts, notably in the test of whether two observed distributions have the same mean. $A(t|\nu)$ is the probability, for ν degrees of freedom, that a certain statistic t (measuring the observed difference of means) would be smaller than the observed value if the means were in fact the same. (See Chapter 14 for further details.) Two means are

significantly different if, e.g., $A(t|\nu) > 0.99$. In other words, $1 - A(t|\nu)$ is the significance level at which the hypothesis that the means are equal is disproved.

The mathematical definition of the function is

$$A(t|\nu) = \frac{1}{\nu^{1/2} B(\frac{1}{2}, \frac{\nu}{2})} \int_{-t}^{t} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}} dx \tag{6.4.7}$$

Limiting values are

$$A(0|\nu) = 0 \qquad A(\infty|\nu) = 1 \tag{6.4.8}$$

$A(t|\nu)$ is related to the incomplete beta function $I_x(a, b)$ by

$$A(t|\nu) = 1 - I_{\frac{\nu}{\nu+t^2}}\left(\frac{\nu}{2}, \frac{1}{2}\right) \tag{6.4.9}$$

So, you can use (6.4.9) and the above routine `betai` to evaluate the function.

F-Distribution Probability Function

This function occurs in the statistical test of whether two observed samples have the same variance. A certain statistic F, essentially the ratio of the observed dispersion of the first sample to that of the second one, is calculated. (For further details, see Chapter 14.) The probability that F would be as *large* as it is if the first sample's underlying distribution actually has *smaller* variance than the second's is denoted $Q(F|\nu_1, \nu_2)$, where ν_1 and ν_2 are the number of degrees of freedom in the first and second samples, respectively. In other words, $Q(F|\nu_1, \nu_2)$ is the significance level at which the hypothesis "1 has smaller variance than 2" can be rejected. A small numerical value implies a very significant rejection, in turn implying high confidence in the hypothesis "1 has variance greater or equal to 2."

$Q(F|\nu_1, \nu_2)$ has the limiting values

$$Q(0|\nu_1, \nu_2) = 1 \qquad Q(\infty|\nu_1, \nu_2) = 0 \tag{6.4.10}$$

Its relation to the incomplete beta function $I_x(a, b)$ as evaluated by `betai` above is

$$Q(F|\nu_1, \nu_2) = I_{\frac{\nu_2}{\nu_2 + \nu_1 F}}\left(\frac{\nu_2}{2}, \frac{\nu_1}{2}\right) \tag{6.4.11}$$

Cumulative Binomial Probability Distribution

Suppose an event occurs with probability p per trial. Then the probability P of its occurring k *or more* times in n trials is termed a *cumulative binomial probability*, and is related to the incomplete beta function $I_x(a, b)$ as follows:

$$P \equiv \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} = I_p(k, n-k+1) \tag{6.4.12}$$

For n larger than a dozen or so, `betai` is a much better way to evaluate the sum in (6.4.12) than would be the straightforward sum with concurrent computation of the binomial coefficients. (For n smaller than a dozen, either method is acceptable.)

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 6 and 26.

Pearson, E., and Johnson, N. 1968, *Tables of the Incomplete Beta Function* (Cambridge: Cambridge University Press).

6.5 Bessel Functions of Integer Order

This section and the next one present practical algorithms for computing various kinds of Bessel functions of integer order. In §6.7 we deal with fractional order. In fact, the more complicated routines for fractional order work fine for integer order too. For integer order, however, the routines in this section (and §6.6) are simpler and faster. Their only drawback is that they are limited by the precision of the underlying rational approximations. For full double precision, it is best to work with the routines for fractional order in §6.7.

For any real ν, the Bessel function $J_\nu(x)$ can be defined by the series representation

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!\Gamma(\nu+k+1)} \tag{6.5.1}$$

The series converges for all x, but it is not computationally very useful for $x \gg 1$.

For ν *not* an integer the Bessel function $Y_\nu(x)$ is given by

$$Y_\nu(x) = \frac{J_\nu(x)\cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)} \tag{6.5.2}$$

The right-hand side goes to the correct limiting value $Y_n(x)$ as ν goes to some integer n, but this is also not computationally useful.

For arguments $x < \nu$, both Bessel functions look qualitatively like simple power laws, with the asymptotic forms for $0 < x \ll \nu$

$$\begin{aligned} J_\nu(x) &\sim \frac{1}{\Gamma(\nu+1)}\left(\frac{1}{2}x\right)^\nu \qquad \nu \geq 0 \\ Y_0(x) &\sim \frac{2}{\pi}\ln(x) \\ Y_\nu(x) &\sim -\frac{\Gamma(\nu)}{\pi}\left(\frac{1}{2}x\right)^{-\nu} \qquad \nu > 0 \end{aligned} \tag{6.5.3}$$

For n larger than a dozen or so, `betai` is a much better way to evaluate the sum in (6.4.12) than would be the straightforward sum with concurrent computation of the binomial coefficients. (For n smaller than a dozen, either method is acceptable.)

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 6 and 26.

Pearson, E., and Johnson, N. 1968, *Tables of the Incomplete Beta Function* (Cambridge: Cambridge University Press).

6.5 Bessel Functions of Integer Order

This section and the next one present practical algorithms for computing various kinds of Bessel functions of integer order. In §6.7 we deal with fractional order. In fact, the more complicated routines for fractional order work fine for integer order too. For integer order, however, the routines in this section (and §6.6) are simpler and faster. Their only drawback is that they are limited by the precision of the underlying rational approximations. For full double precision, it is best to work with the routines for fractional order in §6.7.

For any real ν, the Bessel function $J_\nu(x)$ can be defined by the series representation

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!\Gamma(\nu+k+1)} \tag{6.5.1}$$

The series converges for all x, but it is not computationally very useful for $x \gg 1$.

For ν *not* an integer the Bessel function $Y_\nu(x)$ is given by

$$Y_\nu(x) = \frac{J_\nu(x)\cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)} \tag{6.5.2}$$

The right-hand side goes to the correct limiting value $Y_n(x)$ as ν goes to some integer n, but this is also not computationally useful.

For arguments $x < \nu$, both Bessel functions look qualitatively like simple power laws, with the asymptotic forms for $0 < x \ll \nu$

$$\begin{aligned} J_\nu(x) &\sim \frac{1}{\Gamma(\nu+1)}\left(\frac{1}{2}x\right)^\nu \qquad \nu \geq 0 \\ Y_0(x) &\sim \frac{2}{\pi}\ln(x) \\ Y_\nu(x) &\sim -\frac{\Gamma(\nu)}{\pi}\left(\frac{1}{2}x\right)^{-\nu} \qquad \nu > 0 \end{aligned} \tag{6.5.3}$$

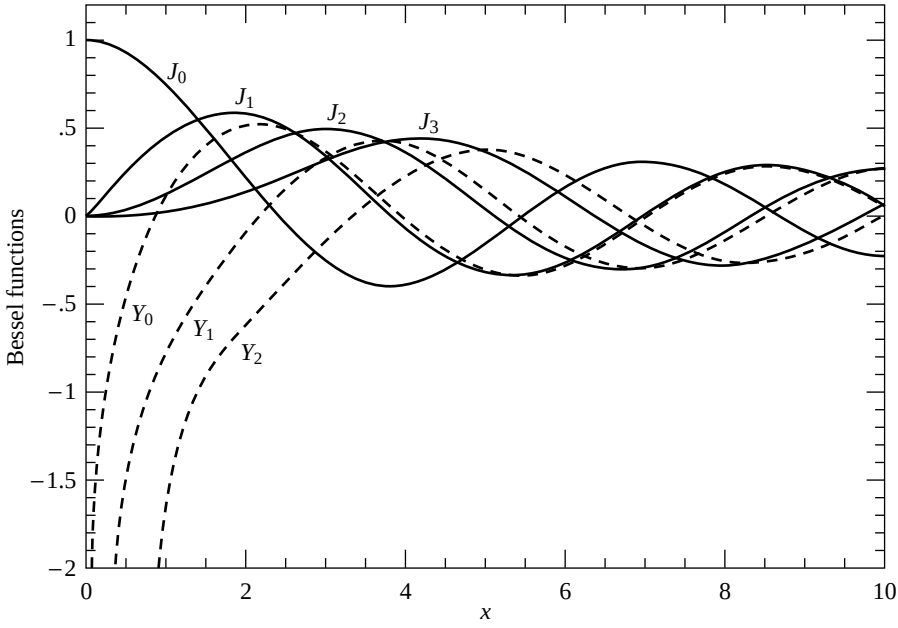


Figure 6.5.1. Bessel functions $J_0(x)$ through $J_3(x)$ and $Y_0(x)$ through $Y_2(x)$.

For $x > \nu$, both Bessel functions look qualitatively like sine or cosine waves whose amplitude decays as $x^{-1/2}$. The asymptotic forms for $x \gg \nu$ are

$$\begin{aligned} J_\nu(x) &\sim \sqrt{\frac{2}{\pi x}}\cos\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \\ Y_\nu(x) &\sim \sqrt{\frac{2}{\pi x}}\sin\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \end{aligned} \tag{6.5.4}$$

In the transition region where $x \sim \nu$, the typical amplitudes of the Bessel functions are on the order

$$\begin{aligned} J_\nu(\nu) &\sim \frac{2^{1/3}}{3^{2/3}\Gamma(\frac{2}{3})}\frac{1}{\nu^{1/3}} \sim \frac{0.4473}{\nu^{1/3}} \\ Y_\nu(\nu) &\sim -\frac{2^{1/3}}{3^{1/6}\Gamma(\frac{2}{3})}\frac{1}{\nu^{1/3}} \sim -\frac{0.7748}{\nu^{1/3}} \end{aligned} \tag{6.5.5}$$

which holds asymptotically for large ν. Figure 6.5.1 plots the first few Bessel functions of each kind.

The Bessel functions satisfy the recurrence relations

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{6.5.6}$$

and

$$Y_{n+1}(x) = \frac{2n}{x}Y_n(x) - Y_{n-1}(x) \tag{6.5.7}$$

As already mentioned in §5.5, only the second of these (6.5.7) is stable in the direction of increasing n for $x < n$. The reason that (6.5.6) is unstable in the

direction of increasing n is simply that it is *the same recurrence* as (6.5.7): A small amount of "polluting" Y_n introduced by roundoff error will quickly come to swamp the desired J_n, according to equation (6.5.3).

A practical strategy for computing the Bessel functions of integer order divides into two tasks: first, how to compute J_0, J_1, Y_0, and Y_1, and second, how to use the recurrence relations stably to find other J's and Y's. We treat the first task first:

For x between zero and some arbitrary value (we will use the value 8), approximate $J_0(x)$ and $J_1(x)$ by rational functions in x. Likewise approximate by rational functions the "regular part" of $Y_0(x)$ and $Y_1(x)$, defined as

$$Y_0(x) - \frac{2}{\pi} J_0(x) \ln(x) \qquad \text{and} \qquad Y_1(x) - \frac{2}{\pi}\left[J_1(x)\ln(x) - \frac{1}{x}\right] \tag{6.5.8}$$

For $8 < x < \infty$, use the approximating forms ($n = 0, 1$)

$$J_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\cos(X_n) - Q_n\left(\frac{8}{x}\right)\sin(X_n)\right] \tag{6.5.9}$$

$$Y_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\sin(X_n) + Q_n\left(\frac{8}{x}\right)\cos(X_n)\right] \tag{6.5.10}$$

where

$$X_n \equiv x - \frac{2n+1}{4}\pi \tag{6.5.11}$$

and where P_0, P_1, Q_0, and Q_1 are each polynomials in their arguments, for $0 < 8/x < 1$. The P's are even polynomials, the Q's odd.

Coefficients of the various rational functions and polynomials are given by Hart [1], for various levels of desired accuracy. A straightforward implementation is

```
      FUNCTION bessj0(x)
      REAL bessj0,x
          Returns the Bessel function J0(x) for any real x.
      REAL ax,xx,z
      DOUBLE PRECISION p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,
     *     r5,r6,s1,s2,s3,s4,s5,s6,y  We'll accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,r5,r6,
     *     s1,s2,s3,s4,s5,s6
      DATA p1,p2,p3,p4,p5/1.d0,-.1098628627d-2,.2734510407d-4,
     *     -.2073370639d-5,.2093887211d-6/, q1,q2,q3,q4,q5/-.1562499995d-1,
     *     .1430488765d-3,-.6911147651d-5,.7621095161d-6,-.934945152d-7/
      DATA r1,r2,r3,r4,r5,r6/57568490574.d0,-13362590354.d0,651619640.7d0,
     *     -11214424.18d0,77392.33017d0,-184.9052456d0/,
     *     s1,s2,s3,s4,s5,s6/57568490411.d0,1029532985.d0,
     *     9494680.718d0,59272.64853d0,267.8532712d0,1.d0/
      if(abs(x).lt.8.)then              Direct rational function fit.
        y=x**2
        bessj0=(r1+y*(r2+y*(r3+y*(r4+y*(r5+y*r6)))))
     *       /(s1+y*(s2+y*(s3+y*(s4+y*(s5+y*s6)))))
      else                              Fitting function (6.5.9).
        ax=abs(x)
        z=8./ax
        y=z**2
        xx=ax-.785398164
        bessj0=sqrt(.636619772/ax)*(cos(xx)*(p1+y*(p2+y*(p3+y*(p4+y
```

```
*          *p5))))-z*sin(xx)*(q1+y*(q2+y*(q3+y*(q4+y*q5)))))
      endif
      return
      END



      FUNCTION bessy0(x)
      REAL bessy0,x
C     USES bessj0
        Returns the Bessel function Y0(x) for positive x.
      REAL xx,z,bessj0
      DOUBLE PRECISION p1,p2,p3,p4,p5,q1,
*       q2,q3,q4,q5,r1,r2,r3,r4,
*       r5,r6,s1,s2,s3,s4,s5,s6,y      We'll accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,
*       r5,r6,s1,s2,s3,s4,s5,s6
      DATA p1,p2,p3,p4,p5/1.d0,-.1098628627d-2,.2734510407d-4,
*       -.2073370639d-5,.2093887211d-6/, q1,q2,q3,q4,q5/-.1562499995d-1,
*       .1430488765d-3,-.6911147651d-5,.7621095161d-6,-.934945152d-7/
      DATA r1,r2,r3,r4,r5,r6/-2957821389.d0,7062834065.d0,-512359803.6d0,
*       10879881.29d0,-86327.92757d0,228.4622733d0/,
*       s1,s2,s3,s4,s5,s6/40076544269.d0,745249964.8d0,
*       7189466.438d0,47447.26470d0,226.1030244d0,1.d0/
      if(x.lt.8.)then                  Rational function approximation of (6.5.8).
        y=x**2
        bessy0=(r1+y*(r2+y*(r3+y*(r4+y*(r5+y*r6)))))/(s1+y*(s2+y
*           *(s3+y*(s4+y*(s5+y*s6)))))+.636619772*bessj0(x)*log(x)
      else                             Fitting function (6.5.10).
        z=8./x
        y=z**2
        xx=x-.785398164
        bessy0=sqrt(.636619772/x)*(sin(xx)*(p1+y*(p2+y*(p3+y*(p4+y*
*           p5))))+z*cos(xx)*(q1+y*(q2+y*(q3+y*(q4+y*q5)))))
      endif
      return
      END



      FUNCTION bessj1(x)
      REAL bessj1,x
        Returns the Bessel function J1(x) for any real x.
      REAL ax,xx,z
      DOUBLE PRECISION p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,
*       r5,r6,s1,s2,s3,s4,s5,s6,y  We'll accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,r5,r6,
*       s1,s2,s3,s4,s5,s6
      DATA r1,r2,r3,r4,r5,r6/72362614232.d0,-7895059235.d0,242396853.1d0,
*       -2972611.439d0,15704.48260d0,-30.16036606d0/,
*       s1,s2,s3,s4,s5,s6/144725228442.d0,2300535178.d0,
*       18583304.74d0,99447.43394d0,376.9991397d0,1.d0/
      DATA p1,p2,p3,p4,p5/1.d0,.183105d-2,-.3516396496d-4,.2457520174d-5,
*       -.240337019d-6/, q1,q2,q3,q4,q5/.04687499995d0,-.2002690873d-3,
*       .8449199096d-5,-.88228987d-6,.105787412d-6/
      if(abs(x).lt.8.)then             Direct rational approximation.
        y=x**2
        bessj1=x*(r1+y*(r2+y*(r3+y*(r4+y*(r5+y*r6)))))
*           /(s1+y*(s2+y*(s3+y*(s4+y*(s5+y*s6)))))
      else                             Fitting function (6.5.9).
        ax=abs(x)
        z=8./ax
        y=z**2
        xx=ax-2.356194491
```

```
        bessj1=sqrt(.636619772/ax)*(cos(xx)*(p1+y*(p2+y*(p3+y*(p4+y
*            *p5))))-z*sin(xx)*(q1+y*(q2+y*(q3+y*(q4+y*q5)))))
*            *sign(1.,x)
      endif
      return
      END


      FUNCTION bessy1(x)
      REAL bessy1,x
C     USES bessj1
        Returns the Bessel function Y_1(x) for positive x.
      REAL xx,z,bessj1
      DOUBLE PRECISION p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,
*       r5,r6,s1,s2,s3,s4,s5,s6,s7,y  We'll accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,q1,q2,q3,q4,q5,r1,r2,r3,r4,
*       r5,r6,s1,s2,s3,s4,s5,s6,s7
      DATA p1,p2,p3,p4,p5/1.d0,.183105d-2,-.3516396496d-4,.2457520174d-5,
*       -.240337019d-6/, q1,q2,q3,q4,q5/.04687499995d0,-.2002690873d-3,
*       .8449199096d-5,-.88228987d-6,.105787412d-6/
      DATA r1,r2,r3,r4,r5,r6/-.4900604943d13,.1275274390d13,-.5153438139d11,
*       .7349264551d9,-.4237922726d7,.8511937935d4/,
*       s1,s2,s3,s4,s5,s6,s7/.2499580570d14,.4244419664d12,
*       .3733650367d10,.2245904002d8,.1020426050d6,.3549632885d3,1.d0/
      if(x.lt.8.)then                 Rational function approximation of (6.5.8).
        y=x**2
        bessy1=x*(r1+y*(r2+y*(r3+y*(r4+y*(r5+y*r6))))/(s1+y*(s2+y*
*            (s3+y*(s4+y*(s5+y*(s6+y*s7))))))+.636619772
*            *(bessj1(x)*log(x)-1./x)
      else                            Fitting function (6.5.10).
        z=8./x
        y=z**2
        xx=x-2.356194491
        bessy1=sqrt(.636619772/x)*(sin(xx)*(p1+y*(p2+y*(p3+y*(p4+y
*            *p5))))+z*cos(xx)*(q1+y*(q2+y*(q3+y*(q4+y*q5)))))
      endif
      return
      END
```

We now turn to the second task, namely how to use the recurrence formulas (6.5.6) and (6.5.7) to get the Bessel functions $J_n(x)$ and $Y_n(x)$ for $n \geq 2$. The latter of these is straightforward, since its upward recurrence is always stable:

```
      FUNCTION bessy(n,x)
      INTEGER n
      REAL bessy,x
C     USES bessy0,bessy1
        Returns the Bessel function Y_n(x) for positive x and n ≥ 2.
      INTEGER j
      REAL by,bym,byp,tox,bessy0,bessy1
      if(n.lt.2)pause 'bad argument n in bessy'
      tox=2./x
      by=bessy1(x)                    Starting values for the recurrence.
      bym=bessy0(x)
      do 11 j=1,n-1                   Recurrence (6.5.7).
        byp=j*tox*by-bym
        bym=by
        by=byp
      enddo 11
      bessy=by
      return
      END
```

The cost of this algorithm is the call to `bessy1` and `bessy0` (which generate a call to each of `bessj1` and `bessj0`), plus $O(\mathtt{n})$ operations in the recurrence.

As for $J_n(x)$, things are a bit more complicated. We can start the recurrence upward on n from J_0 and J_1, but it will remain stable only while n does not exceed x. This is, however, just fine for calls with large x and small n, a case which occurs frequently in practice.

The harder case to provide for is that with $x < n$. The best thing to do here is to use Miller's algorithm (see discussion preceding equation 5.5.16), applying the recurrence *downward* from some arbitrary starting value and making use of the upward-unstable nature of the recurrence to put us *onto* the correct solution. When we finally arrive at J_0 or J_1 we are able to normalize the solution with the sum (5.5.16) accumulated along the way.

The only subtlety is in deciding at how large an n we need start the downward recurrence so as to obtain a desired accuracy by the time we reach the n that we really want. If you play with the asymptotic forms (6.5.3) and (6.5.5), you should be able to convince yourself that the answer is to start larger than the desired n by an additive amount of order $[\text{constant} \times n]^{1/2}$, where the square root of the constant is, very roughly, the number of significant figures of accuracy.

The above considerations lead to the following function.

```
      FUNCTION bessj(n,x)
      INTEGER n,IACC
      REAL bessj,x,BIGNO,BIGNI
      PARAMETER (IACC=40,BIGNO=1.e10,BIGNI=1.e-10)
C     USES bessj0,bessj1
        Returns the Bessel function Jn(x) for any real x and n ≥ 2.
      INTEGER j,jsum,m
      REAL ax,bj,bjm,bjp,sum,tox,bessj0,bessj1
      if(n.lt.2)pause 'bad argument n in bessj'
      ax=abs(x)
      if(ax.eq.0.)then
          bessj=0.
      else if(ax.gt.float(n))then          Upwards recurrence from J0 and J1.
          tox=2./ax
          bjm=bessj0(ax)
          bj=bessj1(ax)
          do 11 j=1,n-1
              bjp=j*tox*bj-bjm
              bjm=bj
              bj=bjp
          enddo 11
          bessj=bj
      else                                 Downwards recurrence from an even m here com-
          tox=2./ax                            puted. Make IACC larger to increase accuracy.
          m=2*((n+int(sqrt(float(IACC*n))))/2)
          bessj=0.
          jsum=0                           jsum will alternate between 0 and 1; when it is 1, we
          sum=0.                               accumulate in sum the even terms in (5.5.16).
          bjp=0.
          bj=1.
          do 12 j=m,1,-1                   The downward recurrence.
              bjm=j*tox*bj-bjp
              bjp=bj
              bj=bjm
              if(abs(bj).gt.BIGNO)then     Renormalize to prevent overflows.
                  bj=bj*BIGNI
```

```
            bjp=bjp*BIGNI
            bessj=bessj*BIGNI
            sum=sum*BIGNI
        endif
        if(jsum.ne.0)sum=sum+bj     Accumulate the sum.
        jsum=1-jsum                 Change 0 to 1 or vice versa.
        if(j.eq.n)bessj=bjp         Save the unnormalized answer.
    enddo 12
    sum=2.*sum-bj                   Compute (5.5.16)
    bessj=bessj/sum                 and use it to normalize the answer.
endif
if(x.lt.0..and.mod(n,2).eq.1)bessj=-bessj
return
END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 9.

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley), §6.8, p. 141. [1]

6.6 Modified Bessel Functions of Integer Order

The modified Bessel functions $I_n(x)$ and $K_n(x)$ are equivalent to the usual Bessel functions J_n and Y_n evaluated for purely imaginary arguments. In detail, the relationship is

$$\begin{aligned} I_n(x) &= (-i)^n J_n(ix) \\ K_n(x) &= \frac{\pi}{2} i^{n+1} [J_n(ix) + iY_n(ix)] \end{aligned} \tag{6.6.1}$$

The particular choice of prefactor and of the linear combination of J_n and Y_n to form K_n are simply choices that make the functions real-valued for real arguments x.

For small arguments $x \ll n$, both $I_n(x)$ and $K_n(x)$ become, asymptotically, simple powers of their argument

$$\begin{aligned} I_n(x) &\approx \frac{1}{n!}\left(\frac{x}{2}\right)^n \qquad n \geq 0 \\ K_0(x) &\approx -\ln(x) \\ K_n(x) &\approx \frac{(n-1)!}{2}\left(\frac{x}{2}\right)^{-n} \qquad n > 0 \end{aligned} \tag{6.6.2}$$

These expressions are virtually identical to those for $J_n(x)$ and $Y_n(x)$ in this region, except for the factor of $-2/\pi$ difference between $Y_n(x)$ and $K_n(x)$. In the region

```
            bjp=bjp*BIGNI
            bessj=bessj*BIGNI
            sum=sum*BIGNI
        endif
        if(jsum.ne.0)sum=sum+bj     Accumulate the sum.
        jsum=1-jsum                 Change 0 to 1 or vice versa.
        if(j.eq.n)bessj=bjp         Save the unnormalized answer.
    enddo 12
    sum=2.*sum-bj                   Compute (5.5.16)
    bessj=bessj/sum                 and use it to normalize the answer.
endif
if(x.lt.0..and.mod(n,2).eq.1)bessj=-bessj
return
END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 9.

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley), §6.8, p. 141. [1]

6.6 Modified Bessel Functions of Integer Order

The modified Bessel functions $I_n(x)$ and $K_n(x)$ are equivalent to the usual Bessel functions J_n and Y_n evaluated for purely imaginary arguments. In detail, the relationship is

$$\begin{aligned} I_n(x) &= (-i)^n J_n(ix) \\ K_n(x) &= \frac{\pi}{2} i^{n+1} [J_n(ix) + iY_n(ix)] \end{aligned} \tag{6.6.1}$$

The particular choice of prefactor and of the linear combination of J_n and Y_n to form K_n are simply choices that make the functions real-valued for real arguments x.

For small arguments $x \ll n$, both $I_n(x)$ and $K_n(x)$ become, asymptotically, simple powers of their argument

$$\begin{aligned} I_n(x) &\approx \frac{1}{n!}\left(\frac{x}{2}\right)^n \qquad n \geq 0 \\ K_0(x) &\approx -\ln(x) \\ K_n(x) &\approx \frac{(n-1)!}{2}\left(\frac{x}{2}\right)^{-n} \qquad n > 0 \end{aligned} \tag{6.6.2}$$

These expressions are virtually identical to those for $J_n(x)$ and $Y_n(x)$ in this region, except for the factor of $-2/\pi$ difference between $Y_n(x)$ and $K_n(x)$. In the region

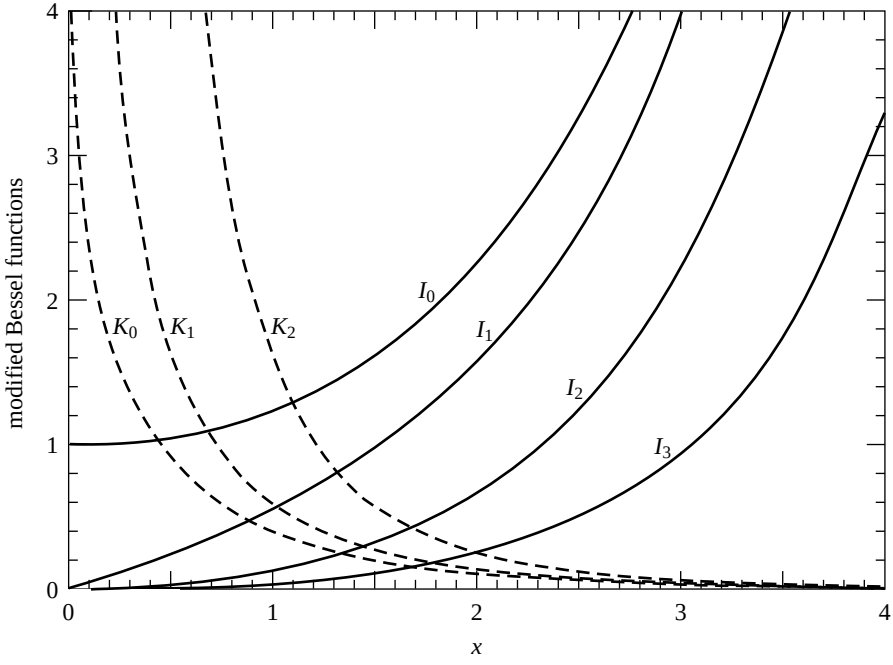


Figure 6.6.1. Modified Bessel functions $I_0(x)$ through $I_3(x)$, $K_0(x)$ through $K_2(x)$.

$x \gg n$, however, the modified functions have quite different behavior than the Bessel functions,

$$I_n(x) \approx \frac{1}{\sqrt{2\pi x}} \exp(x)$$
$$K_n(x) \approx \frac{\pi}{\sqrt{2\pi x}} \exp(-x) \tag{6.6.3}$$

The modified functions evidently have exponential rather than sinusoidal behavior for large arguments (see Figure 6.6.1). The smoothness of the modified Bessel functions, once the exponential factor is removed, makes a simple polynomial approximation of a few terms quite suitable for the functions I_0, I_1, K_0, and K_1. The following routines, based on polynomial coefficients given by Abramowitz and Stegun [1], evaluate these four functions, and will provide the basis for upward recursion for $n > 1$ when $x > n$.

```
      FUNCTION bessi0(x)
      REAL bessi0,x
        Returns the modified Bessel function I0(x) for any real x.
      REAL ax
      DOUBLE PRECISION p1,p2,p3,p4,p5,p6,p7,q1,q2,q3,q4,q5,q6,q7,
     *     q8,q9,y                    Accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,p6,p7,q1,q2,q3,q4,q5,q6,q7,q8,q9
      DATA p1,p2,p3,p4,p5,p6,p7/1.0d0,3.5156229d0,3.0899424d0,1.2067492d0,
     *     0.2659732d0,0.360768d-1,0.45813d-2/
      DATA q1,q2,q3,q4,q5,q6,q7,q8,q9/0.39894228d0,0.1328592d-1,
     *     0.225319d-2,-0.157565d-2,0.916281d-2,-0.2057706d-1,
     *     0.2635537d-1,-0.1647633d-1,0.392377d-2/
```

```
      if (abs(x).lt.3.75) then
          y=(x/3.75)**2
          bessi0=p1+y*(p2+y*(p3+y*(p4+y*(p5+y*(p6+y*p7)))))
      else
          ax=abs(x)
          y=3.75/ax
          bessi0=(exp(ax)/sqrt(ax))*(q1+y*(q2+y*(q3+y*(q4
     *         +y*(q5+y*(q6+y*(q7+y*(q8+y*q9))))))))
      endif
      return
      END



      FUNCTION bessk0(x)
      REAL bessk0,x
C     USES bessi0
        Returns the modified Bessel function K_0(x) for positive real x.
      REAL bessi0
      DOUBLE PRECISION p1,p2,p3,p4,p5,p6,p7,q1,
     *     q2,q3,q4,q5,q6,q7,y                  Accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,p6,p7,q1,q2,q3,q4,q5,q6,q7
      DATA p1,p2,p3,p4,p5,p6,p7/-0.57721566d0,0.42278420d0,0.23069756d0,
     *     0.3488590d-1,0.262698d-2,0.10750d-3,0.74d-5/
      DATA q1,q2,q3,q4,q5,q6,q7/1.25331414d0,-0.7832358d-1,0.2189568d-1,
     *     -0.1062446d-1,0.587872d-2,-0.251540d-2,0.53208d-3/
      if (x.le.2.0) then                        Polynomial fit.
          y=x*x/4.0
          bessk0=(-log(x/2.0)*bessi0(x))+(p1+y*(p2+y*(p3+
     *            y*(p4+y*(p5+y*(p6+y*p7))))))
      else
          y=(2.0/x)
          bessk0=(exp(-x)/sqrt(x))*(q1+y*(q2+y*(q3+
     *            y*(q4+y*(q5+y*(q6+y*q7))))))
      endif
      return
      END



      FUNCTION bessi1(x)
      REAL bessi1,x
        Returns the modified Bessel function I_1(x) for any real x.
      REAL ax
      DOUBLE PRECISION p1,p2,p3,p4,p5,p6,p7,q1,q2,q3,q4,q5,q6,q7,
     *     q8,q9,y                              Accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,p6,p7,q1,q2,q3,q4,q5,q6,q7,q8,q9
      DATA p1,p2,p3,p4,p5,p6,p7/0.5d0,0.87890594d0,0.51498869d0,
     *     0.15084934d0,0.2658733d-1,0.301532d-2,0.32411d-3/
      DATA q1,q2,q3,q4,q5,q6,q7,q8,q9/0.39894228d0,-0.3988024d-1,
     *     -0.362018d-2,0.163801d-2,-0.1031555d-1,0.2282967d-1,
     *     -0.2895312d-1,0.1787654d-1,-0.420059d-2/
      if (abs(x).lt.3.75) then                  Polynomial fit.
          y=(x/3.75)**2
          bessi1=x*(p1+y*(p2+y*(p3+y*(p4+y*(p5+y*(p6+y*p7))))))
      else
          ax=abs(x)
          y=3.75/ax
          bessi1=(exp(ax)/sqrt(ax))*(q1+y*(q2+y*(q3+y*(q4+
     *            y*(q5+y*(q6+y*(q7+y*(q8+y*q9))))))))
          if(x.lt.0.)bessi1=-bessi1
      endif
      return
      END
```

```
      FUNCTION bessk1(x)
      REAL bessk1,x
C     USES bessi1
        Returns the modified Bessel function K1(x) for positive real x.
      REAL bessi1
      DOUBLE PRECISION p1,p2,p3,p4,p5,p6,p7,q1,
     *     q2,q3,q4,q5,q6,q7,y          Accumulate polynomials in double precision.
      SAVE p1,p2,p3,p4,p5,p6,p7,q1,q2,q3,q4,q5,q6,q7
      DATA p1,p2,p3,p4,p5,p6,p7/1.0d0,0.15443144d0,-0.67278579d0,
     *     -0.18156897d0,-0.1919402d-1,-0.110404d-2,-0.4686d-4/
      DATA q1,q2,q3,q4,q5,q6,q7/1.25331414d0,0.23498619d0,-0.3655620d-1,
     *     0.1504268d-1,-0.780353d-2,0.325614d-2,-0.68245d-3/
      if (x.le.2.0) then                Polynomial fit.
        y=x*x/4.0
        bessk1=(log(x/2.0)*bessi1(x))+(1.0/x)*(p1+y*(p2+
     *       y*(p3+y*(p4+y*(p5+y*(p6+y*p7))))))
      else
        y=2.0/x
        bessk1=(exp(-x)/sqrt(x))*(q1+y*(q2+y*(q3+
     *       y*(q4+y*(q5+y*(q6+y*q7))))))
      endif
      return
      END
```

The recurrence relation for $I_n(x)$ and $K_n(x)$ is the same as that for $J_n(x)$ and $Y_n(x)$ provided that ix is substituted for x. This has the effect of changing a sign in the relation,

$$\begin{aligned} I_{n+1}(x) &= -\left(\frac{2n}{x}\right) I_n(x) + I_{n-1}(x) \\ K_{n+1}(x) &= +\left(\frac{2n}{x}\right) K_n(x) + K_{n-1}(x) \end{aligned} \tag{6.6.4}$$

These relations are always *unstable* for upward recurrence. For K_n, itself growing, this presents no problem. For I_n, however, the strategy of downward recursion is therefore required once again, and the starting point for the recursion may be chosen in the same manner as for the routine `bessj`. The only fundamental difference is that the normalization formula for $I_n(x)$ has an alternating minus sign in successive terms, which again arises from the substitution of ix for x in the formula used previously for J_n

$$1 = I_0(x) - 2I_2(x) + 2I_4(x) - 2I_6(x) + \cdots \tag{6.6.5}$$

In fact, we prefer simply to normalize with a call to `bessi0`.

With this simple modification, the recursion routines `bessj` and `bessy` become the new routines `bessi` and `bessk`:

```
      FUNCTION bessk(n,x)
      INTEGER n
      REAL bessk,x
C     USES bessk0,bessk1
        Returns the modified Bessel function Kn(x) for positive x and n ≥ 2.
      INTEGER j
      REAL bk,bkm,bkp,tox,bessk0,bessk1
      if (n.lt.2) pause 'bad argument n in bessk'
      tox=2.0/x
```

```
      bkm=bessk0(x)                    Upward recurrence for all x...
      bk=bessk1(x)
      do 11 j=1,n-1                    ...and here it is.
        bkp=bkm+j*tox*bk
        bkm=bk
        bk=bkp
      enddo 11
      bessk=bk
      return
      END


      FUNCTION bessi(n,x)
      INTEGER n,IACC
      REAL bessi,x,BIGNO,BIGNI
      PARAMETER (IACC=40,BIGNO=1.0e10,BIGNI=1.0e-10)
C     USES bessi0
        Returns the modified Bessel function I_n(x) for any real x and n ≥ 2.
      INTEGER j,m
      REAL bi,bim,bip,tox,bessi0
      if (n.lt.2) pause 'bad argument n in bessi'
      if (x.eq.0.) then
        bessi=0.
      else
        tox=2.0/abs(x)
        bip=0.0
        bi=1.0
        bessi=0.
        m=2*((n+int(sqrt(float(IACC*n))))) Downward recurrence from even m.
        do 11 j=m,1,-1                   Make IACC larger to increase accuracy.
          bim=bip+float(j)*tox*bi        The downward recurrence.
          bip=bi
          bi=bim
          if (abs(bi).gt.BIGNO) then     Renormalize to prevent overflows.
            bessi=bessi*BIGNI
            bi=bi*BIGNI
            bip=bip*BIGNI
          endif
          if (j.eq.n) bessi=bip
        enddo 11
        bessi=bessi*bessi0(x)/bi         Normalize with bessi0.
        if (x.lt.0..and.mod(n,2).eq.1) bessi=-bessi
      endif
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §9.8. [1]

Carrier, G.F., Krook, M. and Pearson, C.E. 1966, *Functions of a Complex Variable* (New York: McGraw-Hill), pp. 220ff.

6.7 Bessel Functions of Fractional Order, Airy Functions, Spherical Bessel Functions

Many algorithms have been proposed for computing Bessel functions of fractional order numerically. Most of them are, in fact, not very good in practice. The routines given here are rather complicated, but they can be recommended wholeheartedly.

Ordinary Bessel Functions

The basic idea is *Steed's method*, which was originally developed [1] for Coulomb wave functions. The method calculates J_ν, J'_ν, Y_ν, and Y'_ν simultaneously, and so involves four relations among these functions. Three of the relations come from two continued fractions, one of which is complex. The fourth is provided by the Wronskian relation

$$W \equiv J_\nu Y'_\nu - Y_\nu J'_\nu = \frac{2}{\pi x} \tag{6.7.1}$$

The first continued fraction, CF1, is defined by

$$\begin{aligned} f_\nu \equiv \frac{J'_\nu}{J_\nu} &= \frac{\nu}{x} - \frac{J_{\nu+1}}{J_\nu} \\ &= \frac{\nu}{x} - \frac{1}{2(\nu+1)/x -} \; \frac{1}{2(\nu+2)/x -} \cdots \end{aligned} \tag{6.7.2}$$

You can easily derive it from the three-term recurrence relation for Bessel functions: Start with equation (6.5.6) and use equation (5.5.18). Forward evaluation of the continued fraction by one of the methods of §5.2 is essentially equivalent to backward recurrence of the recurrence relation. The rate of convergence of CF1 is determined by the position of the *turning point* $x_{\rm tp} = \sqrt{\nu(\nu+1)} \approx \nu$, beyond which the Bessel functions become oscillatory. If $x \lesssim x_{\rm tp}$, convergence is very rapid. If $x \gtrsim x_{\rm tp}$, then each iteration of the continued fraction effectively increases ν by one until $x \lesssim x_{\rm tp}$; thereafter rapid convergence sets in. Thus the number of iterations of CF1 is of order x for large x. In the routine `bessjy` we set the maximum allowed number of iterations to 10,000. For larger x, you can use the usual asymptotic expressions for Bessel functions.

One can show that the sign of J_ν is the same as the sign of the denominator of CF1 once it has converged.

The complex continued fraction CF2 is defined by

$$p + iq \equiv \frac{J'_\nu + iY'_\nu}{J_\nu + iY_\nu} = -\frac{1}{2x} + i + \frac{i}{x} \; \frac{(1/2)^2 - \nu^2}{2(x+i) +} \; \frac{(3/2)^2 - \nu^2}{2(x+2i) +} \cdots \tag{6.7.3}$$

(We sketch the derivation of CF2 in the analogous case of modified Bessel functions in the next subsection.) This continued fraction converges rapidly for $x \gtrsim x_{\rm tp}$, while convergence fails as $x \to 0$. We have to adopt a special method for small x, which we describe below. For x not too small, we can ensure that $x \gtrsim x_{\rm tp}$ by a stable recurrence of J_ν and J'_ν downwards to a value $\nu = \mu \lesssim x$, thus yielding the ratio f_μ at this lower value of ν. This is the stable direction for the recurrence relation. The initial values for the recurrence are

$$J_\nu = \text{arbitrary}, \qquad J'_\nu = f_\nu J_\nu, \tag{6.7.4}$$

with the sign of the arbitrary initial value of J_ν chosen to be the sign of the denominator of CF1. Choosing the initial value of J_ν very small minimizes the possibility of overflow during the recurrence. The recurrence relations are

$$\begin{aligned} J_{\nu-1} &= \frac{\nu}{x} J_\nu + J'_\nu \\ J'_{\nu-1} &= \frac{\nu - 1}{x} J_{\nu-1} - J_\nu \end{aligned} \tag{6.7.5}$$

Once CF2 has been evaluated at $\nu = \mu$, then with the Wronskian (6.7.1) we have enough relations to solve for all four quantities. The formulas are simplified by introducing the quantity

$$\gamma \equiv \frac{p - f_\mu}{q} \tag{6.7.6}$$

Then

$$J_\mu = \pm \left(\frac{W}{q + \gamma(p - f_\mu)} \right)^{1/2} \tag{6.7.7}$$

$$J'_\mu = f_\mu J_\mu \tag{6.7.8}$$

$$Y_\mu = \gamma J_\mu \tag{6.7.9}$$

$$Y'_\mu = Y_\mu \left(p + \frac{q}{\gamma} \right) \tag{6.7.10}$$

The sign of J_μ in (6.7.7) is chosen to be the same as the sign of the initial J_ν in (6.7.4).

Once all four functions have been determined at the value $\nu = \mu$, we can find them at the original value of ν. For J_ν and J'_ν, simply scale the values in (6.7.4) by the ratio of (6.7.7) to the value found after applying the recurrence (6.7.5). The quantities Y_ν and Y'_ν can be found by starting with the values in (6.7.9) and (6.7.10) and using the stable upwards recurrence

$$Y_{\nu+1} = \frac{2\nu}{x} Y_\nu - Y_{\nu-1} \tag{6.7.11}$$

together with the relation

$$Y'_\nu = \frac{\nu}{x} Y_\nu - Y_{\nu+1} \tag{6.7.12}$$

Now turn to the case of small x, when CF2 is not suitable. Temme [2] has given a good method of evaluating Y_ν and $Y_{\nu+1}$, and hence Y'_ν from (6.7.12), by series expansions that accurately handle the singularity as $x \to 0$. The expansions work only for $|\nu| \leq 1/2$, and so now the recurrence (6.7.5) is used to evaluate f_ν at a value $\nu = \mu$ in this interval. Then one calculates J_μ from

$$J_\mu = \frac{W}{Y'_\mu - Y_\mu f_\mu} \tag{6.7.13}$$

and J'_μ from (6.7.8). The values at the original value of ν are determined by scaling as before, and the Y's are recurred up as before.

Temme's series are

$$Y_\nu = -\sum_{k=0}^{\infty} c_k g_k \qquad Y_{\nu+1} = -\frac{2}{x} \sum_{k=0}^{\infty} c_k h_k \tag{6.7.14}$$

Here

$$c_k = \frac{(-x^2/4)^k}{k!} \tag{6.7.15}$$

while the coefficients g_k and h_k are defined in terms of quantities p_k, q_k, and f_k that can be found by recursion:

$$\begin{aligned} g_k &= f_k + \frac{2}{\nu} \sin^2\left(\frac{\nu\pi}{2}\right) q_k \\ h_k &= -k g_k + p_k \\ p_k &= \frac{p_{k-1}}{k - \nu} \\ q_k &= \frac{q_{k-1}}{k + \nu} \\ f_k &= \frac{k f_{k-1} + p_{k-1} + q_{k-1}}{k^2 - \nu^2} \end{aligned} \tag{6.7.16}$$

The initial values for the recurrences are

$$\begin{aligned} p_0 &= \frac{1}{\pi}\left(\frac{x}{2}\right)^{-\nu}\Gamma(1+\nu) \\ q_0 &= \frac{1}{\pi}\left(\frac{x}{2}\right)^{\nu}\Gamma(1-\nu) \\ f_0 &= \frac{2}{\pi}\frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\Gamma_1(\nu)+\frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right] \end{aligned} \tag{6.7.17}$$

with

$$\begin{aligned} \sigma &= \nu\ln\left(\frac{2}{x}\right) \\ \Gamma_1(\nu) &= \frac{1}{2\nu}\left[\frac{1}{\Gamma(1-\nu)}-\frac{1}{\Gamma(1+\nu)}\right] \\ \Gamma_2(\nu) &= \frac{1}{2}\left[\frac{1}{\Gamma(1-\nu)}+\frac{1}{\Gamma(1+\nu)}\right] \end{aligned} \tag{6.7.18}$$

The whole point of writing the formulas in this way is that the potential problems as $\nu \to 0$ can be controlled by evaluating $\nu\pi/\sin\nu\pi$, $\sinh\sigma/\sigma$, and Γ_1 carefully. In particular, Temme gives Chebyshev expansions for $\Gamma_1(\nu)$ and $\Gamma_2(\nu)$. We have rearranged his expansion for Γ_1 to be explicitly an even series in ν so that we can use our routine `chebev` as explained in §5.8.

The routine assumes $\nu \geq 0$. For negative ν you can use the reflection formulas

$$\begin{aligned} J_{-\nu} &= \cos\nu\pi\, J_\nu - \sin\nu\pi\, Y_\nu \\ Y_{-\nu} &= \sin\nu\pi\, J_\nu + \cos\nu\pi\, Y_\nu \end{aligned} \tag{6.7.19}$$

The routine also assumes $x > 0$. For $x < 0$ the functions are in general complex, but expressible in terms of functions with $x > 0$. For $x = 0$, Y_ν is singular.

Internal arithmetic in the routine is carried out in double precision. To maintain portability, complex arithmetic has been recoded with real variables.

```
      SUBROUTINE bessjy(x,xnu,rj,ry,rjp,ryp)
      INTEGER MAXIT
      REAL rj,rjp,ry,ryp,x,xnu,XMIN
      DOUBLE PRECISION EPS,FPMIN,PI
      PARAMETER (EPS=1.e-10,FPMIN=1.e-30,MAXIT=10000,XMIN=2.,
     *      PI=3.141592653589793d0)
C     USES beschb
```

Returns the Bessel functions `rj` $= J_\nu$, `ry` $= Y_\nu$ and their derivatives `rjp` $= J'_\nu$, `ryp` $= Y'_\nu$, for positive `x` and for `xnu` $= \nu \geq 0$. The relative accuracy is within one or two significant digits of EPS, except near a zero of one of the functions, where EPS controls its absolute accuracy. FPMIN is a number close to the machine's smallest floating-point number. All internal arithmetic is in double precision. To convert the entire routine to double precision, change the REAL declaration above and decrease EPS to 10^{-16}. Also convert the subroutine `beschb`.

```
      INTEGER i,isign,l,nl
      DOUBLE PRECISION a,b,br,bi,c,cr,ci,d,del,del1,den,di,dlr,dli,
     *      dr,e,f,fact,fact2,fact3,ff,gam,gam1,gam2,gammi,gampl,h,
     *      p,pimu,pimu2,q,r,rjl,rjl1,rjmu,rjp1,rjpl,rjtemp,ry1,
     *      rymu,rymup,rytemp,sum,sum1,temp,w,x2,xi,xi2,xmu,xmu2
      if(x.le.0..or.xnu.lt.0.) pause 'bad arguments in bessjy'
      if(x.lt.XMIN)then
        nl=int(xnu+.5d0)
      else
        nl=max(0,int(xnu-x+1.5d0))
      endif
      xmu=xnu-nl
      xmu2=xmu*xmu
      xi=1.d0/x
      xi2=2.d0*xi
      w=xi2/PI
```

nl is the number of downward recurrences of the J's and upward recurrences of Y's. xmu lies between $-1/2$ and $1/2$ for x $<$ XMIN, while it is chosen so that x is greater than the turning point for x $\geq$ XMIN.

The Wronskian.

```
      isign=1                          Evaluate CF1 by modified Lentz's method (§5.2). isign keeps
      h=xnu*xi                             track of sign changes in the denominator.
      if(h.lt.FPMIN)h=FPMIN
      b=xi2*xnu
      d=0.d0
      c=h
      do 11 i=1,MAXIT
        b=b+xi2
        d=b-d
        if(abs(d).lt.FPMIN)d=FPMIN
        c=b-1.d0/c
        if(abs(c).lt.FPMIN)c=FPMIN
        d=1.d0/d
        del=c*d
        h=del*h
        if(d.lt.0.d0)isign=-isign
        if(abs(del-1.d0).lt.EPS)goto 1
      enddo 11
      pause 'x too large in bessjy; try asymptotic expansion'
1     continue
      rjl=isign*FPMIN                  Initialize J_ν and J'_ν for downward recurrence.
      rjpl=h*rjl
      rjl1=rjl                         Store values for later rescaling.
      rjp1=rjpl
      fact=xnu*xi
      do 12 l=nl,1,-1
        rjtemp=fact*rjl+rjpl
        fact=fact-xi
        rjpl=fact*rjtemp-rjl
        rjl=rjtemp
      enddo 12
      if(rjl.eq.0.d0)rjl=EPS
      f=rjpl/rjl                       Now have unnormalized J_μ and J'_μ.
      if(x.lt.XMIN) then               Use series.
        x2=.5d0*x
        pimu=PI*xmu
        if(abs(pimu).lt.EPS)then
          fact=1.d0
        else
          fact=pimu/sin(pimu)
        endif
        d=-log(x2)
        e=xmu*d
        if(abs(e).lt.EPS)then
          fact2=1.d0
        else
          fact2=sinh(e)/e
        endif
        call beschb(xmu,gam1,gam2,gampl,gammi)      Chebyshev evaluation of Γ1 and Γ2.
        ff=2.d0/PI*fact*(gam1*cosh(e)+gam2*fact2*d) f_0.
        e=exp(e)
        p=e/(gampl*PI)                              p_0.
        q=1.d0/(e*PI*gammi)                         q_0.
        pimu2=0.5d0*pimu
        if(abs(pimu2).lt.EPS)then
          fact3=1.d0
        else
          fact3=sin(pimu2)/pimu2
        endif
        r=PI*pimu2*fact3*fact3
        c=1.d0
        d=-x2*x2
        sum=ff+r*q
        sum1=p
```

```
        do 13 i=1,MAXIT
            ff=(i*ff+p+q)/(i*i-xmu2)
            c=c*d/i
            p=p/(i-xmu)
            q=q/(i+xmu)
            del=c*(ff+r*q)
            sum=sum+del
            del1=c*p-i*del
            sum1=sum1+del1
            if(abs(del).lt.(1.d0+abs(sum))*EPS)goto 2
        enddo 13
        pause 'bessy series failed to converge'
2       continue
        rymu=-sum
        ry1=-sum1*xi2
        rymup=xmu*xi*rymu-ry1
        rjmu=w/(rymup-f*rymu)                    Equation (6.7.13).
    else                                         Evaluate CF2 by modified Lentz's method
        a=.25d0-xmu2                                 (§5.2).
        p=-.5d0*xi
        q=1.d0
        br=2.d0*x
        bi=2.d0
        fact=a*xi/(p*p+q*q)
        cr=br+q*fact
        ci=bi+p*fact
        den=br*br+bi*bi
        dr=br/den
        di=-bi/den
        dlr=cr*dr-ci*di
        dli=cr*di+ci*dr
        temp=p*dlr-q*dli
        q=p*dli+q*dlr
        p=temp
        do 14 i=2,MAXIT
            a=a+2*(i-1)
            bi=bi+2.d0
            dr=a*dr+br
            di=a*di+bi
            if(abs(dr)+abs(di).lt.FPMIN)dr=FPMIN
            fact=a/(cr*cr+ci*ci)
            cr=br+cr*fact
            ci=bi-ci*fact
            if(abs(cr)+abs(ci).lt.FPMIN)cr=FPMIN
            den=dr*dr+di*di
            dr=dr/den
            di=-di/den
            dlr=cr*dr-ci*di
            dli=cr*di+ci*dr
            temp=p*dlr-q*dli
            q=p*dli+q*dlr
            p=temp
            if(abs(dlr-1.d0)+abs(dli).lt.EPS)goto 3
        enddo 14
        pause 'cf2 failed in bessjy'
3       continue
        gam=(p-f)/q                  Equations (6.7.6) – (6.7.10).
        rjmu=sqrt(w/((p-f)*gam+q))
        rjmu=sign(rjmu,rjl)
        rymu=rjmu*gam
        rymup=rymu*(p+q/gam)
        ry1=xmu*xi*rymu-rymup
    endif
    fact=rjmu/rjl
```

```
      rj=rjl1*fact                    Scale original J_ν and J'_ν.
      rjp=rjp1*fact
      do 15 i=1,nl                    Upward recurrence of Y_ν.
        rytemp=(xmu+i)*xi2*ry1-rymu
        rymu=ry1
        ry1=rytemp
      enddo 15
      ry=rymu
      ryp=xnu*xi*rymu-ry1
      return
      END


      SUBROUTINE beschb(x,gam1,gam2,gampl,gammi)
      INTEGER NUSE1,NUSE2
      DOUBLE PRECISION gam1,gam2,gammi,gampl,x
      PARAMETER (NUSE1=5,NUSE2=5)
C     USES chebev
        Evaluates Γ1 and Γ2 by Chebyshev expansion for |x| ≤ 1/2. Also returns 1/Γ(1 + x) and
        1/Γ(1 − x). If converting to double precision, set NUSE1 = 7, NUSE2 = 8.
      REAL xx,c1(7),c2(8),chebev
      SAVE c1,c2
      DATA c1/-1.142022680371168d0,6.5165112670737d-3,
     *     3.087090173086d-4,-3.4706269649d-6,6.9437664d-9,
     *     3.67795d-11,-1.356d-13/
      DATA c2/1.843740587300905d0,-7.68528408447867d-2,
     *     1.2719271366546d-3,-4.9717367042d-6,-3.31261198d-8,
     *     2.423096d-10,-1.702d-13,-1.49d-15/
      xx=8.d0*x*x-1.d0                Multiply x by 2 to make range be −1 to 1, and then
      gam1=chebev(-1.,1.,c1,NUSE1,xx)   apply transformation for evaluating even Cheby-
      gam2=chebev(-1.,1.,c2,NUSE2,xx)   shev series.
      gampl=gam2-x*gam1
      gammi=gam2+x*gam1
      return
      END
```

Modified Bessel Functions

Steed's method does not work for modified Bessel functions because in this case CF2 is purely imaginary and we have only three relations among the four functions. Temme [3] has given a normalization condition that provides the fourth relation.

The Wronskian relation is

$$W \equiv I_\nu K'_\nu - K_\nu I'_\nu = -\frac{1}{x} \tag{6.7.20}$$

The continued fraction CF1 becomes

$$f_\nu \equiv \frac{I'_\nu}{I_\nu} = \frac{\nu}{x} + \frac{1}{2(\nu+1)/x +} \; \frac{1}{2(\nu+2)/x +} \cdots \tag{6.7.21}$$

To get CF2 and the normalization condition in a convenient form, consider the sequence of confluent hypergeometric functions

$$z_n(x) = U(\nu + 1/2 + n, 2\nu + 1, 2x) \tag{6.7.22}$$

for fixed ν. Then

$$K_\nu(x) = \pi^{1/2}(2x)^\nu e^{-x} z_0(x) \tag{6.7.23}$$

$$\frac{K_{\nu+1}(x)}{K_\nu(x)} = \frac{1}{x}\left[\nu + \frac{1}{2} + x + \left(\nu^2 - \frac{1}{4}\right)\frac{z_1}{z_0}\right] \tag{6.7.24}$$

Equation (6.7.23) is the standard expression for K_ν in terms of a confluent hypergeometric function, while equation (6.7.24) follows from relations between contiguous confluent hypergeometric functions (equations 13.4.16 and 13.4.18 in Abramowitz and Stegun). Now the functions z_n satisfy the three-term recurrence relation (equation 13.4.15 in Abramowitz and Stegun)

$$z_{n-1}(x) = b_n z_n(x) + a_{n+1} z_{n+1} \tag{6.7.25}$$

with

$$\begin{aligned} b_n &= 2(n+x) \\ a_{n+1} &= -[(n+1/2)^2 - \nu^2] \end{aligned} \tag{6.7.26}$$

Following the steps leading to equation (5.5.18), we get the continued fraction CF2

$$\frac{z_1}{z_0} = \frac{1}{b_1 +} \; \frac{a_2}{b_2 +} \cdots \tag{6.7.27}$$

from which (6.7.24) gives $K_{\nu+1}/K_\nu$ and thus K'_ν/K_ν.

Temme's normalization condition is that

$$\sum_{n=0}^{\infty} C_n z_n = \left(\frac{1}{2x}\right)^{\nu+1/2} \tag{6.7.28}$$

where

$$C_n = \frac{(-1)^n}{n!} \frac{\Gamma(\nu + 1/2 + n)}{\Gamma(\nu + 1/2 - n)} \tag{6.7.29}$$

Note that the C_n's can be determined by recursion:

$$C_0 = 1, \qquad C_{n+1} = -\frac{a_{n+1}}{n+1} C_n \tag{6.7.30}$$

We use the condition (6.7.28) by finding

$$S = \sum_{n=1}^{\infty} C_n \frac{z_n}{z_0} \tag{6.7.31}$$

Then

$$z_0 = \left(\frac{1}{2x}\right)^{\nu+1/2} \frac{1}{1+S} \tag{6.7.32}$$

and (6.7.23) gives K_ν.

Thompson and Barnett [4] have given a clever method of doing the sum (6.7.31) simultaneously with the forward evaluation of the continued fraction CF2. Suppose the continued fraction is being evaluated as

$$\frac{z_1}{z_0} = \sum_{n=0}^{\infty} \Delta h_n \tag{6.7.33}$$

where the increments Δh_n are being found by, e.g., Steed's algorithm or the modified Lentz's algorithm of §5.2. Then the approximation to S keeping the first N terms can be found as

$$S_N = \sum_{n=1}^{N} Q_n \Delta h_n \tag{6.7.34}$$

Here

$$Q_n = \sum_{k=1}^{n} C_k q_k \tag{6.7.35}$$

and q_k is found by recursion from

$$q_{k+1} = (q_{k-1} - b_k q_k)/a_{k+1} \tag{6.7.36}$$

starting with $q_0 = 0$, $q_1 = 1$. For the case at hand, approximately three times as many terms are needed to get S to converge as are needed simply for CF2 to converge.

To find K_ν and $K_{\nu+1}$ for small x we use series analogous to (6.7.14):

$$K_\nu = \sum_{k=0}^{\infty} c_k f_k \qquad K_{\nu+1} = \frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \tag{6.7.37}$$

Here

$$\begin{aligned} c_k &= \frac{(x^2/4)^k}{k!} \\ h_k &= -k f_k + p_k \\ p_k &= \frac{p_{k-1}}{k-\nu} \\ q_k &= \frac{q_{k-1}}{k+\nu} \\ f_k &= \frac{k f_{k-1} + p_{k-1} + q_{k-1}}{k^2-\nu^2} \end{aligned} \tag{6.7.38}$$

The initial values for the recurrences are

$$\begin{aligned} p_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{-\nu}\Gamma(1+\nu) \\ q_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{\nu}\Gamma(1-\nu) \\ f_0 &= \frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\Gamma_1(\nu) + \frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right] \end{aligned} \tag{6.7.39}$$

Both the series for small x, and CF2 and the normalization relation (6.7.28) require $|\nu| \le 1/2$. In both cases, therefore, we recurse I_ν down to a value $\nu = \mu$ in this interval, find K_μ there, and recurse K_ν back up to the original value of ν.

The routine assumes $\nu \ge 0$. For negative ν use the reflection formulas

$$\begin{aligned} I_{-\nu} &= I_\nu + \frac{2}{\pi}\sin(\nu\pi)\,K_\nu \\ K_{-\nu} &= K_\nu \end{aligned} \tag{6.7.40}$$

Note that for large x, $I_\nu \sim e^x$, $K_\nu \sim e^{-x}$, and so these functions will overflow or underflow. It is often desirable to be able to compute the scaled quantities $e^{-x}I_\nu$ and $e^x K_\nu$. Simply omitting the factor e^{-x} in equation (6.7.23) will ensure that all four quantities will have the appropriate scaling. If you also want to scale the four quantities for small x when the series in equation (6.7.37) are used, you must multiply each series by e^x.

```
      SUBROUTINE bessik(x,xnu,ri,rk,rip,rkp)
      INTEGER MAXIT
      REAL ri,rip,rk,rkp,x,xnu,XMIN
      DOUBLE PRECISION EPS,FPMIN,PI
      PARAMETER (EPS=1.e-10,FPMIN=1.e-30,MAXIT=10000,XMIN=2.,
     *     PI=3.141592653589793d0)
C     USES beschb
```

Returns the modified Bessel functions `ri` $= I_\nu$, `rk` $= K_\nu$ and their derivatives `rip` $= I'_\nu$, `rkp` $= K'_\nu$, for positive `x` and for `xnu` $= \nu \ge 0$. The relative accuracy is within one or two significant digits of `EPS`. `FPMIN` is a number close to the machine's smallest floating-point number. All internal arithmetic is in double precision. To convert the entire routine to double precision, change the `REAL` declaration above and decrease `EPS` to 10^{-16}. Also convert the subroutine `beschb`.

```
      INTEGER i,l,nl
      DOUBLE PRECISION a,a1,b,c,d,del,del1,delh,dels,e,f,fact,
     *     fact2,ff,gam1,gam2,gammi,gampl,h,p,pimu,q,q1,q2,
     *     qnew,ril,ril1,rimu,rip1,ripl,ritemp,rk1,rkmu,rkmup,
     *     rktemp,s,sum,sum1,x2,xi,xi2,xmu,xmu2
      if(x.le.0..or.xnu.lt.0.) pause 'bad arguments in bessik'
```

```
      nl=int(xnu+.5d0)                                  nl is the number of downward recurrences
      xmu=xnu-nl                                            of the I's and upward recurrences
      xmu2=xmu*xmu                                          of K's. xmu lies between −1/2 and
      xi=1.d0/x                                             1/2.
      xi2=2.d0*xi
      h=xnu*xi                                          Evaluate CF1 by modified Lentz's method
      if(h.lt.FPMIN)h=FPMIN                                 (§5.2).
      b=xi2*xnu
      d=0.d0
      c=h
      do 11 i=1,MAXIT
        b=b+xi2
        d=1.d0/(b+d)                                    Denominators cannot be zero here, so no
        c=b+1.d0/c                                          need for special precautions.
        del=c*d
        h=del*h
        if(abs(del-1.d0).lt.EPS)goto 1
      enddo 11
      pause 'x too large in bessik; try asymptotic expansion'
1     continue
      ril=FPMIN                                         Initialize I_ν and I'_ν for downward recur-
      ripl=h*ril                                            rence.
      ril1=ril                                          Store values for later rescaling.
      rip1=ripl
      fact=xnu*xi
      do 12 l=nl,1,-1
        ritemp=fact*ril+ripl
        fact=fact-xi
        ripl=fact*ritemp+ril
        ril=ritemp
      enddo 12
      f=ripl/ril                                        Now have unnormalized I_μ and I'_μ.
      if(x.lt.XMIN) then                                Use series.
        x2=.5d0*x
        pimu=PI*xmu
        if(abs(pimu).lt.EPS)then
          fact=1.d0
        else
          fact=pimu/sin(pimu)
        endif
        d=-log(x2)
        e=xmu*d
        if(abs(e).lt.EPS)then
          fact2=1.d0
        else
          fact2=sinh(e)/e
        endif
        call beschb(xmu,gam1,gam2,gampl,gammi)          Chebyshev evaluation of Γ1 and Γ2.
        ff=fact*(gam1*cosh(e)+gam2*fact2*d)             f0.
        sum=ff
        e=exp(e)
        p=0.5d0*e/gampl                                 p0.
        q=0.5d0/(e*gammi)                               q0.
        c=1.d0
        d=x2*x2
        sum1=p
        do 13 i=1,MAXIT
          ff=(i*ff+p+q)/(i*i-xmu2)
          c=c*d/i
          p=p/(i-xmu)
          q=q/(i+xmu)
          del=c*ff
          sum=sum+del
          del1=c*(p-i*ff)
```

```
            sum1=sum1+del1
            if(abs(del).lt.abs(sum)*EPS)goto 2
        enddo 13
        pause 'bessk series failed to converge'
2       continue
        rkmu=sum
        rk1=sum1*xi2
    else                                        Evaluate CF2 by Steed's algorithm (§5.2),
        b=2.d0*(1.d0+x)                             which is OK because there can be no
        d=1.d0/b                                    zero denominators.
        delh=d
        h=delh
        q1=0.d0                                 Initializations for recurrence (6.7.35).
        q2=1.d0
        a1=.25d0-xmu2
        c=a1
        q=c                                     First term in equation (6.7.34).
        a=-a1
        s=1.d0+q*delh
        do 14 i=2,MAXIT
            a=a-2*(i-1)
            c=-a*c/i
            qnew=(q1-b*q2)/a
            q1=q2
            q2=qnew
            q=q+c*qnew
            b=b+2.d0
            d=1.d0/(b+a*d)
            delh=(b*d-1.d0)*delh
            h=h+delh
            dels=q*delh
            s=s+dels
            if(abs(dels/s).lt.EPS)goto 3        Need only test convergence of sum since
        enddo 14                                    CF2 itself converges more quickly.
        pause 'bessik: failure to converge in cf2'
3       continue
        h=a1*h
        rkmu=sqrt(PI/(2.d0*x))*exp(-x)/s        Omit the factor exp(−x) to scale all the
        rk1=rkmu*(xmu+x+.5d0-h)*xi                  returned functions by exp(x) for x ≥
    endif                                           XMIN.
    rkmup=xmu*xi*rkmu-rk1
    rimu=xi/(f*rkmu-rkmup)                      Get Iμ from Wronskian.
    ri=(rimu*ril1)/ril                          Scale original Iν and I′ν.
    rip=(rimu*rip1)/ril
    do 15 i=1,nl                                Upward recurrence of Kν.
        rktemp=(xmu+i)*xi2*rk1+rkmu
        rkmu=rk1
        rk1=rktemp
    enddo 15
    rk=rkmu
    rkp=xnu*xi*rkmu-rk1
    return
    END
```

Airy Functions

For positive x, the Airy functions are defined by

$$\mathrm{Ai}(x) = \frac{1}{\pi}\sqrt{\frac{x}{3}}K_{1/3}(z) \tag{6.7.41}$$

$$\mathrm{Bi}(x) = \sqrt{\frac{x}{3}}[I_{1/3}(z) + I_{-1/3}(z)] \tag{6.7.42}$$

where

$$z = \frac{2}{3}x^{3/2} \tag{6.7.43}$$

By using the reflection formula (6.7.40), we can convert (6.7.42) into the computationally more useful form

$$\mathrm{Bi}(x) = \sqrt{x}\left[\frac{2}{\sqrt{3}}I_{1/3}(z) + \frac{1}{\pi}K_{1/3}(z)\right] \tag{6.7.44}$$

so that Ai and Bi can be evaluated with a single call to `bessik`.

The derivatives should not be evaluated by simply differentiating the above expressions because of possible subtraction errors near $x = 0$. Instead, use the equivalent expressions

$$\begin{aligned} \mathrm{Ai}'(x) &= -\frac{x}{\pi\sqrt{3}}K_{2/3}(z) \\ \mathrm{Bi}'(x) &= x\left[\frac{2}{\sqrt{3}}I_{2/3}(z) + \frac{1}{\pi}K_{2/3}(z)\right] \end{aligned} \tag{6.7.45}$$

The corresponding formulas for negative arguments are

$$\begin{aligned} \mathrm{Ai}(-x) &= \frac{\sqrt{x}}{2}\left[J_{1/3}(z) - \frac{1}{\sqrt{3}}Y_{1/3}(z)\right] \\ \mathrm{Bi}(-x) &= -\frac{\sqrt{x}}{2}\left[\frac{1}{\sqrt{3}}J_{1/3}(z) + Y_{1/3}(z)\right] \\ \mathrm{Ai}'(-x) &= \frac{x}{2}\left[J_{2/3}(z) + \frac{1}{\sqrt{3}}Y_{2/3}(z)\right] \\ \mathrm{Bi}'(-x) &= \frac{x}{2}\left[\frac{1}{\sqrt{3}}J_{2/3}(z) - Y_{2/3}(z)\right] \end{aligned} \tag{6.7.46}$$

```
      SUBROUTINE airy(x,ai,bi,aip,bip)
      REAL ai,aip,bi,bip,x
C     USES bessik,bessjy
        Returns Airy functions Ai(x), Bi(x), and their derivatives Ai'(x), Bi'(x).
      REAL absx,ri,rip,rj,rjp,rk,rkp,rootx,ry,ryp,z,
     *     PI,THIRD,TWOTHR,ONOVRT
      PARAMETER (PI=3.1415927,THIRD=1./3.,TWOTHR=2.*THIRD,
     *     ONOVRT=.57735027)
      absx=abs(x)
      rootx=sqrt(absx)
      z=TWOTHR*absx*rootx
      if(x.gt.0.)then
        call bessik(z,THIRD,ri,rk,rip,rkp)
        ai=rootx*ONOVRT*rk/PI
        bi=rootx*(rk/PI+2.*ONOVRT*ri)
        call bessik(z,TWOTHR,ri,rk,rip,rkp)
        aip=-x*ONOVRT*rk/PI
        bip=x*(rk/PI+2.*ONOVRT*ri)
      else if(x.lt.0.)then
        call bessjy(z,THIRD,rj,ry,rjp,ryp)
        ai=.5*rootx*(rj-ONOVRT*ry)
        bi=-.5*rootx*(ry+ONOVRT*rj)
        call bessjy(z,TWOTHR,rj,ry,rjp,ryp)
        aip=.5*absx*(ONOVRT*ry+rj)
        bip=.5*absx*(ONOVRT*rj-ry)
      else                                        Case x = 0.
        ai=.35502805
        bi=ai/ONOVRT
```

```
    aip=-.25881940
    bip=-aip/ONOVRT
endif
return
END
```

Spherical Bessel Functions

For integer n, spherical Bessel functions are defined by

$$j_n(x) = \sqrt{\frac{\pi}{2x}} J_{n+(1/2)}(x)$$
$$y_n(x) = \sqrt{\frac{\pi}{2x}} Y_{n+(1/2)}(x) \tag{6.7.47}$$

They can be evaluated by a call to `bessjy`, and the derivatives can safely be found from the derivatives of equation (6.7.47).

Note that in the continued fraction CF2 in (6.7.3) just the first term survives for $\nu = 1/2$. Thus one can make a very simple algorithm for spherical Bessel functions along the lines of `bessjy` by always recursing j_n down to $n = 0$, setting p and q from the first term in CF2, and then recursing y_n up. No special series is required near $x = 0$. However, `bessjy` is already so efficient that we have not bothered to provide an independent routine for spherical Bessels.

```
SUBROUTINE sphbes(n,x,sj,sy,sjp,syp)
INTEGER n
REAL sj,sjp,sy,syp,x
C   USES bessjy
    Returns spherical Bessel functions jn(x), yn(x), and their derivatives jn'(x), yn'(x) for
    integer n.
REAL factor,order,rj,rjp,ry,ryp,RTPIO2
PARAMETER (RTPIO2=1.2533141)
if(n.lt.0.or.x.le.0.)pause 'bad arguments in sphbes'
order=n+0.5
call bessjy(x,order,rj,ry,rjp,ryp)
factor=RTPIO2/sqrt(x)
sj=factor*rj
sy=factor*ry
sjp=factor*rjp-sj/(2.*x)
syp=factor*ryp-sy/(2.*x)
return
END
```

CITED REFERENCES AND FURTHER READING:

Barnett, A.R., Feng, D.H., Steed, J.W., and Goldfarb, L.J.B. 1974, *Computer Physics Communications*, vol. 8, pp. 377–395. [1]

Temme, N.M. 1976, *Journal of Computational Physics*, vol. 21, pp. 343–350 [2]; 1975, *op. cit.*, vol. 19, pp. 324–337. [3]

Thompson, I.J., and Barnett, A.R. 1987, *Computer Physics Communications*, vol. 47, pp. 245–257. [4]

Barnett, A.R. 1981, *Computer Physics Communications*, vol. 21, pp. 297–314.

Thompson, I.J., and Barnett, A.R. 1986, *Journal of Computational Physics*, vol. 64, pp. 490–509.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 10.

6.8 Spherical Harmonics

Spherical harmonics occur in a large variety of physical problems, for example, whenever a wave equation, or Laplace's equation, is solved by separation of variables in spherical coordinates. The spherical harmonic $Y_{lm}(\theta,\phi)$, $-l \le m \le l$, is a function of the two coordinates θ, ϕ on the surface of a sphere.

The spherical harmonics are orthogonal for different l and m, and they are normalized so that their integrated square over the sphere is unity:

$$\int_0^{2\pi} d\phi \int_{-1}^{1} d(\cos\theta) Y_{l'm'}{}^*(\theta,\phi) Y_{lm}(\theta,\phi) = \delta_{l'l}\delta_{m'm} \tag{6.8.1}$$

Here asterisk denotes complex conjugation.

Mathematically, the spherical harmonics are related to *associated Legendre polynomials* by the equation

$$Y_{lm}(\theta,\phi) = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta) e^{im\phi} \tag{6.8.2}$$

By using the relation

$$Y_{l,-m}(\theta,\phi) = (-1)^m Y_{lm}{}^*(\theta,\phi) \tag{6.8.3}$$

we can always relate a spherical harmonic to an associated Legendre polynomial with $m \ge 0$. With $x \equiv \cos\theta$, these are defined in terms of the ordinary Legendre polynomials (cf. §4.5 and §5.5) by

$$P_l^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x) \tag{6.8.4}$$

The first few associated Legendre polynomials, and their corresponding normalized spherical harmonics, are

$P_0^0(x) = 1$	$Y_{00} = \sqrt{\frac{1}{4\pi}}$
$P_1^1(x) = -(1-x^2)^{1/2}$	$Y_{11} = -\sqrt{\frac{3}{8\pi}}\sin\theta e^{i\phi}$
$P_1^0(x) = x$	$Y_{10} = \sqrt{\frac{3}{4\pi}}\cos\theta$
$P_2^2(x) = 3(1-x^2)$	$Y_{22} = \frac{1}{4}\sqrt{\frac{15}{2\pi}}\sin^2\theta e^{2i\phi}$
$P_2^1(x) = -3(1-x^2)^{1/2}x$	$Y_{21} = -\sqrt{\frac{15}{8\pi}}\sin\theta\cos\theta e^{i\phi}$
$P_2^0(x) = \frac{1}{2}(3x^2-1)$	$Y_{20} = \sqrt{\frac{5}{4\pi}}(\frac{3}{2}\cos^2\theta - \frac{1}{2})$

(6.8.5)

There are many bad ways to evaluate associated Legendre polynomials numerically. For example, there are explicit expressions, such as

$$P_l^m(x) = \frac{(-1)^m (l+m)!}{2^m m!(l-m)!}(1-x^2)^{m/2}\left[1 - \frac{(l-m)(m+l+1)}{1!(m+1)}\left(\frac{1-x}{2}\right) + \frac{(l-m)(l-m-1)(m+l+1)(m+l+2)}{2!(m+1)(m+2)}\left(\frac{1-x}{2}\right)^2 - \cdots\right] \tag{6.8.6}$$

where the polynomial continues up through the term in $(1-x)^{l-m}$. (See [1] for this and related formulas.) This is not a satisfactory method because evaluation of the polynomial involves delicate cancellations between successive terms, which alternate in sign. For large l, the individual terms in the polynomial become very much larger than their sum, and all accuracy is lost.

In practice, (6.8.6) can be used only in single precision (32-bit) for l up to 6 or 8, and in double precision (64-bit) for l up to 15 or 18, depending on the precision required for the answer. A more robust computational procedure is therefore desirable, as follows:

The associated Legendre functions satisfy numerous recurrence relations, tabulated in [1-2]. These are recurrences on l alone, on m alone, and on both l and m simultaneously. Most of the recurrences involving m are unstable, and so dangerous for numerical work. The following recurrence on l is, however, stable (compare 5.5.1):

$$(l-m)P_l^m = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m \tag{6.8.7}$$

It is useful because there is a closed-form expression for the starting value,

$$P_m^m = (-1)^m(2m-1)!!(1-x^2)^{m/2} \tag{6.8.8}$$

(The notation $n!!$ denotes the product of all *odd* integers less than or equal to n.) Using (6.8.7) with $l = m+1$, and setting $P_{m-1}^m = 0$, we find

$$P_{m+1}^m = x(2m+1)P_m^m \tag{6.8.9}$$

Equations (6.8.8) and (6.8.9) provide the two starting values required for (6.8.7) for general l.

The function that implements this is

```
FUNCTION plgndr(l,m,x)
INTEGER l,m
REAL plgndr,x
    Computes the associated Legendre polynomial P_l^m(x). Here m and l are integers satisfying
    0 ≤ m ≤ l, while x lies in the range −1 ≤ x ≤ 1.
INTEGER i,ll
REAL fact,pll,pmm,pmmp1,somx2
if(m.lt.0.or.m.gt.l.or.abs(x).gt.1.)pause 'bad arguments in plgndr'
pmm=1.                              Compute P_m^m.
if(m.gt.0) then
    somx2=sqrt((1.-x)*(1.+x))
    fact=1.
    do 11 i=1,m
        pmm=-pmm*fact*somx2
        fact=fact+2.
    enddo 11
endif
if(l.eq.m) then
    plgndr=pmm
else
    pmmp1=x*(2*m+1)*pmm             Compute P_{m+1}^m.
    if(l.eq.m+1) then
        plgndr=pmmp1
    else                            Compute P_l^m, l > m + 1.
        do 12 ll=m+2,l
```

```
            pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m)
            pmm=pmmp1
            pmmp1=pll
          enddo 12
          plgndr=pll
        endif
      endif
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea), pp. 54ff. [1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 8. [2]

6.9 Fresnel Integrals, Cosine and Sine Integrals

Fresnel Integrals

The two Fresnel integrals are defined by

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right) dt, \qquad S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right) dt \tag{6.9.1}$$

The most convenient way of evaluating these functions to arbitrary precision is to use power series for small x and a continued fraction for large x. The series are

$$\begin{aligned} C(x) &= x - \left(\frac{\pi}{2}\right)^2 \frac{x^5}{5\cdot 2!} + \left(\frac{\pi}{2}\right)^4 \frac{x^9}{9\cdot 4!} - \cdots \\ S(x) &= \left(\frac{\pi}{2}\right) \frac{x^3}{3\cdot 1!} - \left(\frac{\pi}{2}\right)^3 \frac{x^7}{7\cdot 3!} + \left(\frac{\pi}{2}\right)^5 \frac{x^{11}}{11\cdot 5!} - \cdots \end{aligned} \tag{6.9.2}$$

There is a complex continued fraction that yields both $S(x)$ and $C(x)$ simultaneously:

$$C(x) + iS(x) = \frac{1+i}{2} \operatorname{erf} z, \qquad z = \frac{\sqrt{\pi}}{2}(1-i)x \tag{6.9.3}$$

where

$$\begin{aligned} e^{z^2} \operatorname{erfc} z &= \frac{1}{\sqrt{\pi}} \left(\frac{1}{z+} \, \frac{1/2}{z+} \, \frac{1}{z+} \, \frac{3/2}{z+} \, \frac{2}{z+} \cdots \right) \\ &= \frac{2z}{\sqrt{\pi}} \left(\frac{1}{2z^2+1-} \, \frac{1\cdot 2}{2z^2+5-} \, \frac{3\cdot 4}{2z^2+9-} \cdots \right) \end{aligned} \tag{6.9.4}$$

```
            pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m)
            pmm=pmmp1
            pmmp1=pll
          enddo 12
          plgndr=pll
        endif
      endif
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea), pp. 54ff. [1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 8. [2]

6.9 Fresnel Integrals, Cosine and Sine Integrals

Fresnel Integrals

The two Fresnel integrals are defined by

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right) dt, \qquad S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right) dt \tag{6.9.1}$$

The most convenient way of evaluating these functions to arbitrary precision is to use power series for small x and a continued fraction for large x. The series are

$$\begin{aligned} C(x) &= x - \left(\frac{\pi}{2}\right)^2 \frac{x^5}{5\cdot 2!} + \left(\frac{\pi}{2}\right)^4 \frac{x^9}{9\cdot 4!} - \cdots \\ S(x) &= \left(\frac{\pi}{2}\right) \frac{x^3}{3\cdot 1!} - \left(\frac{\pi}{2}\right)^3 \frac{x^7}{7\cdot 3!} + \left(\frac{\pi}{2}\right)^5 \frac{x^{11}}{11\cdot 5!} - \cdots \end{aligned} \tag{6.9.2}$$

There is a complex continued fraction that yields both $S(x)$ and $C(x)$ simultaneously:

$$C(x) + iS(x) = \frac{1+i}{2} \operatorname{erf} z, \qquad z = \frac{\sqrt{\pi}}{2}(1-i)x \tag{6.9.3}$$

where

$$\begin{aligned} e^{z^2} \operatorname{erfc} z &= \frac{1}{\sqrt{\pi}} \left(\frac{1}{z+} \, \frac{1/2}{z+} \, \frac{1}{z+} \, \frac{3/2}{z+} \, \frac{2}{z+} \cdots \right) \\ &= \frac{2z}{\sqrt{\pi}} \left(\frac{1}{2z^2+1-} \, \frac{1\cdot 2}{2z^2+5-} \, \frac{3\cdot 4}{2z^2+9-} \cdots \right) \end{aligned} \tag{6.9.4}$$

In the last line we have converted the "standard" form of the continued fraction to its "even" form (see §5.2), which converges twice as fast. We must be careful not to evaluate the alternating series (6.9.2) at too large a value of x; inspection of the terms shows that $x = 1.5$ is a good point to switch over to the continued fraction.

Note that for large x

$$C(x) \sim \frac{1}{2} + \frac{1}{\pi x}\sin\left(\frac{\pi}{2}x^2\right), \qquad S(x) \sim \frac{1}{2} - \frac{1}{\pi x}\cos\left(\frac{\pi}{2}x^2\right) \tag{6.9.5}$$

Thus the precision of the routine `frenel` may be limited by the precision of the library routines for sine and cosine for large x.

```
      SUBROUTINE frenel(x,s,c)
      INTEGER MAXIT
      REAL c,s,x,EPS,FPMIN,PI,PIBY2,XMIN
      PARAMETER (EPS=6.e-8,MAXIT=100,FPMIN=1.e-30,XMIN=1.5,
     *     PI=3.1415927,PIBY2=1.5707963)
```

Computes the Fresnel integrals $S(x)$ and $C(x)$ for all real x.

Parameters: EPS is the relative error; MAXIT is the maximum number of iterations allowed; FPMIN is a number near the smallest representable floating-point number; XMIN is the dividing line between using the series and continued fraction; PI $= \pi$; PIBY2 $= \pi/2$.

```
      INTEGER k,n
      REAL a,absc,ax,fact,pix2,sign,sum,sumc,sums,term,test
      COMPLEX b,cc,d,h,del,cs
      LOGICAL odd
      absc(h)=abs(real(h))+abs(aimag(h))      Statement function.
      ax=abs(x)
      if(ax.lt.sqrt(FPMIN))then               Special case: avoid failure of convergence test
          s=0.                                    because of underflow.
          c=ax
      else if(ax.le.XMIN)then                 Evaluate both series simultaneously.
          sum=0.
          sums=0.
          sumc=ax
          sign=1.
          fact=PIBY2*ax*ax
          odd=.true.
          term=ax
          n=3
          do 11 k=1,MAXIT
              term=term*fact/k
              sum=sum+sign*term/n
              test=abs(sum)*EPS
              if(odd)then
                  sign=-sign
                  sums=sum
                  sum=sumc
              else
                  sumc=sum
                  sum=sums
              endif
              if(term.lt.test)goto 1
              odd=.not.odd
              n=n+2
          enddo 11
          pause 'series failed in frenel'
1         s=sums
          c=sumc
      else                                    Evaluate continued fraction by modified Lentz's
          pix2=PI*ax*ax                           method (§5.2).
          b=cmplx(1.,-pix2)
```

```
        cc=1./FPMIN
        d=1./b
        h=d
        n=-1
        do 12 k=2,MAXIT
            n=n+2
            a=-n*(n+1)
            b=b+4.
            d=1./(a*d+b)                    Denominators cannot be zero.
            cc=b+a/cc
            del=cc*d
            h=h*del
            if(absc(del-1.).lt.EPS)goto 2
        enddo 12
        pause 'cf failed in frenel'
2       h=h*cmplx(ax,-ax)
        cs=cmplx(.5,.5)*(1.-cmplx(cos(.5*pix2),sin(.5*pix2))*h)
        c=real(cs)
        s=aimag(cs)
    endif
    if(x.lt.0.)then                         Use antisymmetry.
        c=-c
        s=-s
    endif
    return
    END
```

Cosine and Sine Integrals

The cosine and sine integrals are defined by

$$\begin{aligned} \mathrm{Ci}(x) &= \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t}\, dt \\ \mathrm{Si}(x) &= \int_0^x \frac{\sin t}{t}\, dt \end{aligned} \tag{6.9.6}$$

Here $\gamma \approx 0.5772\ldots$ is Euler's constant. We only need a way to calculate the functions for $x > 0$, because

$$\mathrm{Si}(-x) = -\,\mathrm{Si}(x), \qquad \mathrm{Ci}(-x) = \mathrm{Ci}(x) - i\pi \tag{6.9.7}$$

Once again we can evaluate these functions by a judicious combination of power series and complex continued fraction. The series are

$$\begin{aligned} \mathrm{Si}(x) &= x - \frac{x^3}{3 \cdot 3!} + \frac{x^5}{5 \cdot 5!} - \cdots \\ \mathrm{Ci}(x) &= \gamma + \ln x + \left(-\frac{x^2}{2 \cdot 2!} + \frac{x^4}{4 \cdot 4!} - \cdots \right) \end{aligned} \tag{6.9.8}$$

The continued fraction for the exponential integral $E_1(ix)$ is

$$\begin{aligned} E_1(ix) &= -\,\mathrm{Ci}(x) + i[\mathrm{Si}(x) - \pi/2] \\ &= e^{-ix} \left(\frac{1}{ix+}\, \frac{1}{1+}\, \frac{1}{ix+}\, \frac{2}{1+}\, \frac{2}{ix+} \cdots \right) \\ &= e^{-ix} \left(\frac{1}{1+ix-}\, \frac{1^2}{3+ix-}\, \frac{2^2}{5+ix-} \cdots \right) \end{aligned} \tag{6.9.9}$$

The "even" form of the continued fraction is given in the last line and converges twice as fast for about the same amount of computation. A good crossover point from the alternating series to the continued fraction is $x = 2$ in this case. As for the Fresnel integrals, for large x the precision may be limited by the precision of the sine and cosine routines.

```
      SUBROUTINE cisi(x,ci,si)
      INTEGER MAXIT
      REAL ci,si,x,EPS,EULER,PIBY2,FPMIN,TMIN
      PARAMETER (EPS=6.e-8,EULER=.57721566,MAXIT=100,PIBY2=1.5707963,
     *     FPMIN=1.e-30,TMIN=2.)
```

Computes the cosine and sine integrals $\mathrm{Ci}(x)$ and $\mathrm{Si}(x)$. $\mathrm{Ci}(0)$ is returned as a large negative number and no error message is generated. For $x < 0$ the routine returns $\mathrm{Ci}(-x)$ and you must supply the $-i\pi$ yourself.

Parameters: EPS is the relative error, or absolute error near a zero of $\mathrm{Ci}(x)$; EULER $= \gamma$; MAXIT is the maximum number of iterations allowed; PIBY2 $= \pi/2$; FPMIN is a number near the smallest representable floating-point number; TMIN is the dividing line between using the series and continued fraction.

```
      INTEGER i,k
      REAL a,err,fact,sign,sum,sumc,sums,t,term,absc
      COMPLEX h,b,c,d,del
      LOGICAL odd
      absc(h)=abs(real(h))+abs(aimag(h))      Statement function.
      t=abs(x)
      if(t.eq.0.)then                         Special case.
        si=0.
        ci=-1./FPMIN
        return
      endif
      if(t.gt.TMIN)then                       Evaluate continued fraction by modified Lentz's
        b=cmplx(1.,t)                           method (§5.2).
        c=1./FPMIN
        d=1./b
        h=d
        do 11 i=2,MAXIT
          a=-(i-1)**2
          b=b+2.
          d=1./(a*d+b)                        Denominators cannot be zero.
          c=b+a/c
          del=c*d
          h=h*del
          if(abs(del-1.).lt.EPS)goto 1
        enddo 11
        pause 'cf failed in cisi'
1       continue
        h=cmplx(cos(t),-sin(t))*h
        ci=-real(h)
        si=PIBY2+aimag(h)
      else                                    Evaluate both series simultaneously.
        if(t.lt.sqrt(FPMIN))then              Special case: avoid failure of convergence test
          sumc=0.                               because of underflow.
          sums=t
        else
          sum=0.
          sums=0.
          sumc=0.
          sign=1.
          fact=1.
          odd=.true.
          do 12 k=1,MAXIT
            fact=fact*t/k
            term=fact/k
```

```
            sum=sum+sign*term
            err=term/abs(sum)
            if(odd)then
                sign=-sign
                sums=sum
                sum=sumc
            else
                sumc=sum
                sum=sums
            endif
            if(err.lt.EPS)goto 2
            odd=.not.odd
          enddo 12
          pause 'maxits exceeded in cisi'
        endif
2       si=sums
        ci=sumc+log(t)+EULER
      endif
      if(x.lt.0.)si=-si
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Stegun, I.A., and Zucker, R. 1976, *Journal of Research of the National Bureau of Standards*, vol. 80B, pp. 291–311; 1981, *op. cit.*, vol. 86, pp. 661–686.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 5 and 7.

6.10 Dawson's Integral

Dawson's Integral $F(x)$ is defined by

$$F(x) = e^{-x^2} \int_0^x e^{t^2}\, dt \tag{6.10.1}$$

The function can also be related to the complex error function by

$$F(z) = \frac{i\sqrt{\pi}}{2} e^{-z^2} \left[1 - \mathrm{erfc}(-iz)\right]. \tag{6.10.2}$$

A remarkable approximation for $F(x)$, due to Rybicki [1], is

$$F(z) = \lim_{h \to 0} \frac{1}{\sqrt{\pi}} \sum_{n \text{ odd}} \frac{e^{-(z-nh)^2}}{n} \tag{6.10.3}$$

What makes equation (6.10.3) unusual is that its accuracy increases *exponentially* as h gets small, so that quite moderate values of h (and correspondingly quite rapid convergence of the series) give very accurate approximations.

```
            sum=sum+sign*term
            err=term/abs(sum)
            if(odd)then
                sign=-sign
                sums=sum
                sum=sumc
            else
                sumc=sum
                sum=sums
            endif
            if(err.lt.EPS)goto 2
            odd=.not.odd
          enddo 12
          pause 'maxits exceeded in cisi'
        endif
2       si=sums
        ci=sumc+log(t)+EULER
      endif
      if(x.lt.0.)si=-si
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Stegun, I.A., and Zucker, R. 1976, *Journal of Research of the National Bureau of Standards*, vol. 80B, pp. 291–311; 1981, *op. cit.*, vol. 86, pp. 661–686.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 5 and 7.

6.10 Dawson's Integral

Dawson's Integral $F(x)$ is defined by

$$F(x) = e^{-x^2} \int_0^x e^{t^2}\, dt \tag{6.10.1}$$

The function can also be related to the complex error function by

$$F(z) = \frac{i\sqrt{\pi}}{2} e^{-z^2} \left[1 - \mathrm{erfc}(-iz)\right]. \tag{6.10.2}$$

A remarkable approximation for $F(x)$, due to Rybicki [1], is

$$F(z) = \lim_{h \to 0} \frac{1}{\sqrt{\pi}} \sum_{n \text{ odd}} \frac{e^{-(z-nh)^2}}{n} \tag{6.10.3}$$

What makes equation (6.10.3) unusual is that its accuracy increases *exponentially* as h gets small, so that quite moderate values of h (and correspondingly quite rapid convergence of the series) give very accurate approximations.

We will discuss the theory that leads to equation (6.10.3) later, in §13.11, as an interesting application of Fourier methods. Here we simply implement a routine based on the formula.

It is first convenient to shift the summation index to center it approximately on the maximum of the exponential term. Define n_0 to be the even integer nearest to x/h, and $x_0 \equiv n_0 h$, $x' \equiv x - x_0$, and $n' \equiv n - n_0$, so that

$$F(x) \approx \frac{1}{\sqrt{\pi}} \sum_{\substack{n'=-N \\ n' \text{ odd}}}^{N} \frac{e^{-(x'-n'h)^2}}{n'+n_0}, \tag{6.10.4}$$

where the approximate equality is accurate when h is sufficiently small and N is sufficiently large. The computation of this formula can be greatly speeded up if we note that

$$e^{-(x'-n'h)^2} = e^{-x'^2} e^{-(n'h)^2} \left(e^{2x'h}\right)^{n'}. \tag{6.10.5}$$

The first factor is computed once, the second is an array of constants to be stored, and the third can be computed recursively, so that only two exponentials need be evaluated. Advantage is also taken of the symmetry of the coefficients $e^{-(n'h)^2}$ by breaking the summation up into positive and negative values of n' separately.

In the following routine, the choices $h = 0.4$ and $N = 11$ are made. Because of the symmetry of the summations and the restriction to odd values of n, the limits on the do loops are 1 to 6. The accuracy of the result in this REAL version is about 2×10^{-7}. In order to maintain relative accuracy near $x = 0$, where $F(x)$ vanishes, the program branches to the evaluation of the power series [2] for $F(x)$, for $|x| < 0.2$.

```
FUNCTION dawson(x)
INTEGER NMAX
REAL dawson,x,H,A1,A2,A3
PARAMETER (NMAX=6,H=0.4,A1=2./3.,A2=0.4,A3=2./7.)
    Returns Dawson's integral F(x) = exp(-x^2) int_0^x exp(t^2)dt for any real x.
INTEGER i,init,n0
REAL d1,d2,e1,e2,sum,x2,xp,xx,c(NMAX)
SAVE init,c
DATA init/0/                    Flag is 0 if we need to initialize, else 1.
if(init.eq.0)then
    init=1
    do 11 i=1,NMAX
        c(i)=exp(-((2.*float(i)-1.)*H)**2)
    enddo 11
endif
if(abs(x).lt.0.2)then           Use series expansion.
    x2=x**2
    dawson=x*(1.-A1*x2*(1.-A2*x2*(1.-A3*x2)))
else                            Use sampling theorem representation.
    xx=abs(x)
    n0=2*nint(0.5*xx/H)
    xp=xx-float(n0)*H
    e1=exp(2.*xp*H)
    e2=e1**2
    d1=float(n0+1)
    d2=d1-2.
    sum=0.
    do 12 i=1,NMAX
```

```
        sum=sum+c(i)*(e1/d1+1./(d2*e1))
        d1=d1+2.
        d2=d2-2.
        e1=e2*e1
    enddo 12
    dawson=0.5641895835*sign(exp(-xp**2),x)*sum      Constant is 1/√π.
endif
return
END
```

Other methods for computing Dawson's integral are also known [2,3].

CITED REFERENCES AND FURTHER READING:

Rybicki, G.B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85–87. [1]

Cody, W.J., Pociorek, K.A., and Thatcher, H.C. 1970, *Mathematics of Computation*, vol. 24, pp. 171–178. [2]

McCabe, J.H. 1974, *Mathematics of Computation*, vol. 28, pp. 811–816. [3]

6.11 Elliptic Integrals and Jacobian Elliptic Functions

Elliptic integrals occur in many applications, because any integral of the form

$$\int R(t, s)\, dt \tag{6.11.1}$$

where R is a rational function of t and s, and s is the square root of a cubic or quartic polynomial in t, can be evaluated in terms of elliptic integrals. Standard references [1] describe how to carry out the reduction, which was originally done by Legendre. Legendre showed that only three basic elliptic integrals are required. The simplest of these is

$$I_1 = \int_y^x \frac{dt}{\sqrt{(a_1 + b_1 t)(a_2 + b_2 t)(a_3 + b_3 t)(a_4 + b_4 t)}} \tag{6.11.2}$$

where we have written the quartic s^2 in factored form. In standard integral tables [2], one of the limits of integration is always a zero of the quartic, while the other limit lies closer than the next zero, so that there is no singularity within the interval. To evaluate I_1, we simply break the interval $[y, x]$ into subintervals, each of which either begins or ends on a singularity. The tables, therefore, need only distinguish the eight cases in which each of the four zeros (ordered according to size) appears as the upper or lower limit of integration. In addition, when one of the b's in (6.11.2) tends to zero, the quartic reduces to a cubic, with the largest or smallest singularity moving to $\pm\infty$; this leads to eight more cases (actually just special cases of the first eight). The sixteen cases in total are then usually tabulated in terms of Legendre's standard elliptic integral of the 1st kind, which we will define below. By a change of the variable of integration t, the zeros of the quartic are mapped to standard locations

```
        sum=sum+c(i)*(e1/d1+1./(d2*e1))
        d1=d1+2.
        d2=d2-2.
        e1=e2*e1
    enddo 12
    dawson=0.5641895835*sign(exp(-xp**2),x)*sum      Constant is 1/√π.
endif
return
END
```

Other methods for computing Dawson's integral are also known [2,3].

CITED REFERENCES AND FURTHER READING:

Rybicki, G.B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85–87. [1]

Cody, W.J., Pociorek, K.A., and Thatcher, H.C. 1970, *Mathematics of Computation*, vol. 24, pp. 171–178. [2]

McCabe, J.H. 1974, *Mathematics of Computation*, vol. 28, pp. 811–816. [3]

6.11 Elliptic Integrals and Jacobian Elliptic Functions

Elliptic integrals occur in many applications, because any integral of the form

$$\int R(t, s)\, dt \tag{6.11.1}$$

where R is a rational function of t and s, and s is the square root of a cubic or quartic polynomial in t, can be evaluated in terms of elliptic integrals. Standard references [1] describe how to carry out the reduction, which was originally done by Legendre. Legendre showed that only three basic elliptic integrals are required. The simplest of these is

$$I_1 = \int_y^x \frac{dt}{\sqrt{(a_1 + b_1 t)(a_2 + b_2 t)(a_3 + b_3 t)(a_4 + b_4 t)}} \tag{6.11.2}$$

where we have written the quartic s^2 in factored form. In standard integral tables [2], one of the limits of integration is always a zero of the quartic, while the other limit lies closer than the next zero, so that there is no singularity within the interval. To evaluate I_1, we simply break the interval $[y, x]$ into subintervals, each of which either begins or ends on a singularity. The tables, therefore, need only distinguish the eight cases in which each of the four zeros (ordered according to size) appears as the upper or lower limit of integration. In addition, when one of the b's in (6.11.2) tends to zero, the quartic reduces to a cubic, with the largest or smallest singularity moving to $\pm\infty$; this leads to eight more cases (actually just special cases of the first eight). The sixteen cases in total are then usually tabulated in terms of Legendre's standard elliptic integral of the 1st kind, which we will define below. By a change of the variable of integration t, the zeros of the quartic are mapped to standard locations

on the real axis. Then only two dimensionless parameters are needed to tabulate Legendre's integral. However, the symmetry of the original integral (6.11.2) under permutation of the roots is concealed in Legendre's notation. We will get back to Legendre's notation below. But first, here is a better way:

Carlson [3] has given a new definition of a standard elliptic integral of the first kind,

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}} \tag{6.11.3}$$

where x, y, and z are nonnegative and at most one is zero. By standardizing the range of integration, he retains permutation symmetry for the zeros. (Weierstrass' canonical form also has this property.) Carlson first shows that when x or y is a zero of the quartic in (6.11.2), the integral I_1 can be written in terms of R_F in a form that is symmetric under permutation of the *remaining* three zeros. In the general case when neither x nor y is a zero, two such R_F functions can be combined into a single one by an *addition theorem*, leading to the fundamental formula

$$I_1 = 2R_F(U_{12}^2, U_{13}^2, U_{14}^2) \tag{6.11.4}$$

where

$$U_{ij} = (X_iX_jY_kY_m + Y_iY_jX_kX_m)/(x-y) \tag{6.11.5}$$

$$X_i = (a_i + b_ix)^{1/2}, \qquad Y_i = (a_i + b_iy)^{1/2} \tag{6.11.6}$$

and i, j, k, m is any permutation of $1, 2, 3, 4$. A short-cut in evaluating these expressions is

$$\begin{aligned} U_{13}^2 &= U_{12}^2 - (a_1b_4 - a_4b_1)(a_2b_3 - a_3b_2) \\ U_{14}^2 &= U_{12}^2 - (a_1b_3 - a_3b_1)(a_2b_4 - a_4b_2) \end{aligned} \tag{6.11.7}$$

The U's correspond to the three ways of pairing the four zeros, and I_1 is thus manifestly symmetric under permutation of the zeros. Equation (6.11.4) therefore reproduces all sixteen cases when one limit is a zero, and also includes the cases when neither limit is a zero.

Thus Carlson's function allows arbitrary ranges of integration and arbitrary positions of the branch points of the integrand relative to the interval of integration. To handle elliptic integrals of the second and third kind, Carlson defines the standard integral of the third kind as

$$R_J(x,y,z,p) = \frac{3}{2}\int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}} \tag{6.11.8}$$

which is symmetric in x, y, and z. The degenerate case when two arguments are equal is denoted

$$R_D(x,y,z) = R_J(x,y,z,z) \tag{6.11.9}$$

and is symmetric in x and y. The function R_D replaces Legendre's integral of the second kind. The degenerate form of R_F is denoted

$$R_C(x,y) = R_F(x,y,y) \tag{6.11.10}$$

It embraces logarithmic, inverse circular, and inverse hyperbolic functions.

Carlson [4-7] gives integral tables in terms of the exponents of the linear factors of the quartic in (6.11.1). For example, the integral where the exponents are $(\frac{1}{2},\frac{1}{2},-\frac{1}{2},-\frac{3}{2})$ can be expressed as a single integral in terms of R_D; it accounts for 144 separate cases in Gradshteyn and Ryzhik [2]!

Refer to Carlson's papers [3-7] for some of the practical details in reducing elliptic integrals to his standard forms, such as handling complex conjugate zeros.

Turn now to the numerical evaluation of elliptic integrals. The traditional methods [8] are Gauss or Landen transformations. *Descending* transformations decrease the modulus k of the Legendre integrals towards zero, *increasing* transformations increase it towards unity. In these limits the functions have simple analytic expressions. While these methods converge quadratically and are quite satisfactory for integrals of the first and second kinds, they generally lead to loss of significant figures in certain regimes for integrals of the third kind. Carlson's algorithms [9,10], by contrast, provide a unified method for all three kinds with no significant cancellations.

The key ingredient in these algorithms is the *duplication theorem*:

$$\begin{aligned} R_F(x,y,z) &= 2R_F(x+\lambda, y+\lambda, z+\lambda) \\ &= R_F\left(\frac{x+\lambda}{4}, \frac{y+\lambda}{4}, \frac{z+\lambda}{4}\right) \end{aligned} \tag{6.11.11}$$

where

$$\lambda = (xy)^{1/2} + (xz)^{1/2} + (yz)^{1/2} \tag{6.11.12}$$

This theorem can be proved by a simple change of variable of integration [11]. Equation (6.11.11) is iterated until the arguments of R_F are nearly equal. For equal arguments we have

$$R_F(x,x,x) = x^{-1/2} \tag{6.11.13}$$

When the arguments are close enough, the function is evaluated from a fixed Taylor expansion about (6.11.13) through fifth-order terms. While the iterative part of the algorithm is only linearly convergent, the error ultimately decreases by a factor of $4^6 = 4096$ for each iteration. Typically only two or three iterations are required, perhaps six or seven if the initial values of the arguments have huge ratios. We list the algorithm for R_F here, and refer you to Carlson's paper [9] for the other cases.

Stage 1: For $n = 0, 1, 2, \ldots$ compute

$$\mu_n = (x_n + y_n + z_n)/3$$

$$X_n = 1 - (x_n/\mu_n), \quad Y_n = 1 - (y_n/\mu_n), \quad Z_n = 1 - (z_n/\mu_n)$$

$$\epsilon_n = \max(|X_n|, |Y_n|, |Z_n|)$$

If $\epsilon_n < \text{tol}$ go to Stage 2; else compute

$$\lambda_n = (x_n y_n)^{1/2} + (x_n z_n)^{1/2} + (y_n z_n)^{1/2}$$

$$x_{n+1} = (x_n + \lambda_n)/4, \quad y_{n+1} = (y_n + \lambda_n)/4, \quad z_{n+1} = (z_n + \lambda_n)/4$$

and repeat this stage.

Stage 2: Compute

$$E_2 = X_n Y_n - Z_n^2, \quad E_3 = X_n Y_n Z_n$$

$$R_F = (1 - \tfrac{1}{10}E_2 + \tfrac{1}{14}E_3 + \tfrac{1}{24}E_2^2 - \tfrac{3}{44}E_2 E_3)/(\mu_n)^{1/2}$$

In some applications the argument p in R_J or the argument y in R_C is negative, and the Cauchy principal value of the integral is required. This is easily handled by using the formulas

$$\begin{aligned} &R_J(x,y,z,p) = \\ &\qquad [(\gamma - y)R_J(x,y,z,\gamma) - 3R_F(x,y,z) + 3R_C(xz/y, p\gamma/y)]\,/(y-p) \end{aligned} \tag{6.11.14}$$

where

$$\gamma \equiv y + \frac{(z-y)(y-x)}{y-p} \tag{6.11.15}$$

is positive if p is negative, and

$$R_C(x,y) = \left(\frac{x}{x-y}\right)^{1/2} R_C(x-y,-y) \tag{6.11.16}$$

The Cauchy principal value of R_J has a zero at some value of $p < 0$, so (6.11.14) will give some loss of significant figures near the zero.

```
      FUNCTION rf(x,y,z)
      REAL rf,x,y,z,ERRTOL,TINY,BIG,THIRD,C1,C2,C3,C4
      PARAMETER (ERRTOL=.08,TINY=1.5e-38,BIG=3.E37,THIRD=1./3.,
     *     C1=1./24.,C2=.1,C3=3./44.,C4=1./14.)
```

Computes Carlson's elliptic integral of the first kind, $R_F(x,y,z)$. x, y, and z must be nonnegative, and at most one can be zero. TINY must be at least 5 times the machine underflow limit, BIG at most one fifth the machine overflow limit.

```
      REAL alamb,ave,delx,dely,delz,e2,e3,sqrtx,sqrty,sqrtz,xt,yt,zt
      if(min(x,y,z).lt.0..or.min(x+y,x+z,y+z).lt.TINY.or.
     *     max(x,y,z).gt.BIG)pause 'invalid arguments in rf'
      xt=x
      yt=y
      zt=z
1     continue
          sqrtx=sqrt(xt)
          sqrty=sqrt(yt)
          sqrtz=sqrt(zt)
          alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
          xt=.25*(xt+alamb)
          yt=.25*(yt+alamb)
          zt=.25*(zt+alamb)
          ave=THIRD*(xt+yt+zt)
          delx=(ave-xt)/ave
          dely=(ave-yt)/ave
          delz=(ave-zt)/ave
      if(max(abs(delx),abs(dely),abs(delz)).gt.ERRTOL)goto 1
      e2=delx*dely-delz**2
      e3=delx*dely*delz
      rf=(1.+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave)
      return
      END
```

A value of 0.08 for the error tolerance parameter is adequate for single precision (7 significant digits). Since the error scales as ϵ_n^6, we see that 0.0025 will yield double precision (16 significant digits) and require at most two or three more iterations. Since the coefficients of the sixth-order truncation error are different for the other elliptic functions, these values for the error tolerance should be changed to 0.04 and 0.0012 in the algorithm for R_C, and 0.05 and 0.0015 for R_J and R_D. As well as being an algorithm in its own right for certain combinations of elementary functions, the algorithm for R_C is used repeatedly in the computation of R_J.

The Fortran implementations test the input arguments against two machine-dependent constants, TINY and BIG, to ensure that there will be no underflow or overflow during the computation. We have chosen conservative values, corresponding to a machine minimum of 3×10^{-39} and a machine maximum of 1.7×10^{38}. You can always extend the range of admissible argument values by using the homogeneity relations (6.11.22), below.

```
      FUNCTION rd(x,y,z)
      REAL rd,x,y,z,ERRTOL,TINY,BIG,C1,C2,C3,C4,C5,C6
      PARAMETER (ERRTOL=.05,TINY=1.e-25,BIG=4.5E21,C1=3./14.,C2=1./6.,
     *     C3=9./22.,C4=3./26.,C5=.25*C3,C6=1.5*C4)
```

Computes Carlson's elliptic integral of the second kind, $R_D(x,y,z)$. x and y must be nonnegative, and at most one can be zero. z must be positive. TINY must be at least twice the negative 2/3 power of the machine overflow limit. BIG must be at most $0.1 \times$ ERRTOL times the negative 2/3 power of the machine underflow limit.

```
      REAL alamb,ave,delx,dely,delz,ea,eb,ec,ed,ee,fac,sqrtx,sqrty,
```

```
*       sqrtz,sum,xt,yt,zt
      if(min(x,y).lt.0..or.min(x+y,z).lt.TINY.or.
*       max(x,y,z).gt.BIG)pause 'invalid arguments in rd'
      xt=x
      yt=y
      zt=z
      sum=0.
      fac=1.
1     continue
          sqrtx=sqrt(xt)
          sqrty=sqrt(yt)
          sqrtz=sqrt(zt)
          alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
          sum=sum+fac/(sqrtz*(zt+alamb))
          fac=.25*fac
          xt=.25*(xt+alamb)
          yt=.25*(yt+alamb)
          zt=.25*(zt+alamb)
          ave=.2*(xt+yt+3.*zt)
          delx=(ave-xt)/ave
          dely=(ave-yt)/ave
          delz=(ave-zt)/ave
      if(max(abs(delx),abs(dely),abs(delz)).gt.ERRTOL)goto 1
      ea=delx*dely
      eb=delz*delz
      ec=ea-eb
      ed=ea-6.*eb
      ee=ed+ec+ec
      rd=3.*sum+fac*(1.+ed*(-C1+C5*ed-C6*delz*ee)
*       +delz*(C2*ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave))
      return
      END



      FUNCTION rj(x,y,z,p)
      REAL rj,p,x,y,z,ERRTOL,TINY,BIG,C1,C2,C3,C4,C5,C6,C7,C8
      PARAMETER (ERRTOL=.05,TINY=2.5e-13,BIG=9.E11,C1=3./14.,C2=1./3.,
*       C3=3./22.,C4=3./26.,C5=.75*C3,C6=1.5*C4,C7=.5*C2,C8=C3+C3)
C     USES rc,rf
```

Computes Carlson's elliptic integral of the third kind, $R_J(x, y, z, p)$. x, y, and z must be nonnegative, and at most one can be zero. p must be nonzero. If $p < 0$, the Cauchy principal value is returned. TINY must be at least twice the cube root of the machine underflow limit, BIG at most one fifth the cube root of the machine overflow limit.

```
      REAL a,alamb,alpha,ave,b,beta,delp,delx,dely,delz,ea,eb,ec,
*       ed,ee,fac,pt,rcx,rho,sqrtx,sqrty,sqrtz,sum,tau,xt,
*       yt,zt,rc,rf
      if(min(x,y,z).lt.0..or.min(x+y,x+z,y+z,abs(p)).lt.TINY.or.
*       max(x,y,z,abs(p)).gt.BIG)pause 'invalid arguments in rj'
      sum=0.
      fac=1.
      if(p.gt.0.)then
          xt=x
          yt=y
          zt=z
          pt=p
      else
          xt=min(x,y,z)
          zt=max(x,y,z)
          yt=x+y+z-xt-zt
          a=1./(yt-p)
          b=a*(zt-yt)*(yt-xt)
          pt=yt+b
          rho=xt*zt/yt
```

```
        tau=p*pt/yt
        rcx=rc(rho,tau)
      endif
1     continue
        sqrtx=sqrt(xt)
        sqrty=sqrt(yt)
        sqrtz=sqrt(zt)
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
        alpha=(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz)**2
        beta=pt*(pt+alamb)**2
        sum=sum+fac*rc(alpha,beta)
        fac=.25*fac
        xt=.25*(xt+alamb)
        yt=.25*(yt+alamb)
        zt=.25*(zt+alamb)
        pt=.25*(pt+alamb)
        ave=.2*(xt+yt+zt+pt+pt)
        delx=(ave-xt)/ave
        dely=(ave-yt)/ave
        delz=(ave-zt)/ave
        delp=(ave-pt)/ave
      if(max(abs(delx),abs(dely),abs(delz),abs(delp)).gt.ERRTOL)goto 1
      ea=delx*(dely+delz)+dely*delz
      eb=delx*dely*delz
      ec=delp**2
      ed=ea-3.*ec
      ee=eb+2.*delp*(ea-ec)
      rj=3.*sum+fac*(1.+ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8+delp*C4))
*          +delp*ea*(C2-delp*C3)-C2*delp*ec)/(ave*sqrt(ave))
      if (p.le.0.) rj=a*(b*rj+3.*(rcx-rf(xt,yt,zt)))
      return
      END



      FUNCTION rc(x,y)
      REAL rc,x,y,ERRTOL,TINY,SQRTNY,BIG,TNBG,COMP1,COMP2,THIRD,
*          C1,C2,C3,C4
      PARAMETER (ERRTOL=.04,TINY=1.69e-38,SQRTNY=1.3e-19,BIG=3.E37,
*          TNBG=TINY*BIG,COMP1=2.236/SQRTNY,COMP2=TNBG*TNBG/25.,
*          THIRD=1./3.,C1=.3,C2=1./7.,C3=.375,C4=9./22.)
```

Computes Carlson's degenerate elliptic integral, $R_C(x, y)$. x must be nonnegative and y must be nonzero. If $y < 0$, the Cauchy principal value is returned. TINY must be at least 5 times the machine underflow limit, BIG at most one fifth the machine maximum overflow limit.

```
      REAL alamb,ave,s,w,xt,yt
      if(x.lt.0..or.y.eq.0..or.(x+abs(y)).lt.TINY.or.(x+abs(y)).gt.BIG
*          .or.(y.lt.-COMP1.and.x.gt.0..and.x.lt.COMP2))
*          pause 'invalid arguments in rc'
      if(y.gt.0.)then
        xt=x
        yt=y
        w=1.
      else
        xt=x-y
        yt=-y
        w=sqrt(x)/sqrt(xt)
      endif
1     continue
        alamb=2.*sqrt(xt)*sqrt(yt)+yt
        xt=.25*(xt+alamb)
        yt=.25*(yt+alamb)
        ave=THIRD*(xt+yt+yt)
        s=(yt-ave)/ave
```

```
if(abs(s).gt.ERRTOL)goto 1
rc=w*(1.+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave)
return
END
```

At times you may want to express your answer in Legendre's notation. Alternatively, you may be given results in that notation and need to compute their values with the programs given above. It is a simple matter to transform back and forth. The *Legendre elliptic integral of the 1st kind* is defined as

$$F(\phi,k)\equiv\int_0^\phi \frac{d\theta}{\sqrt{1-k^2\sin^2\theta}} \tag{6.11.17}$$

The *complete elliptic integral of the 1st kind* is given by

$$K(k)\equiv F(\pi/2,k) \tag{6.11.18}$$

In terms of R_F,

$$\begin{aligned} F(\phi,k) &= \sin\phi R_F(\cos^2\phi, 1-k^2\sin^2\phi, 1) \\ K(k) &= R_F(0,1-k^2,1) \end{aligned} \tag{6.11.19}$$

The *Legendre elliptic integral of the 2nd kind* and the *complete elliptic integral of the 2nd kind* are given by

$$\begin{aligned} E(\phi,k) &\equiv \int_0^\phi \sqrt{1-k^2\sin^2\theta}\,d\theta \\ &= \sin\phi R_F(\cos^2\phi,1-k^2\sin^2\phi,1) \\ &\quad -\tfrac{1}{3}k^2\sin^3\phi R_D(\cos^2\phi,1-k^2\sin^2\phi,1) \\ E(k) &\equiv E(\pi/2,k) = R_F(0,1-k^2,1)-\tfrac{1}{3}k^2R_D(0,1-k^2,1) \end{aligned} \tag{6.11.20}$$

Finally, the *Legendre elliptic integral of the 3rd kind* is

$$\begin{aligned} \Pi(\phi,n,k) &\equiv \int_0^\phi \frac{d\theta}{(1+n\sin^2\theta)\sqrt{1-k^2\sin^2\theta}} \\ &= \sin\phi R_F(\cos^2\phi,1-k^2\sin^2\phi,1) \\ &\quad -\tfrac{1}{3}n\sin^3\phi R_J(\cos^2\phi,1-k^2\sin^2\phi,1,1+n\sin^2\phi) \end{aligned} \tag{6.11.21}$$

(Note that this sign convention for n is opposite that of Abramowitz and Stegun [12], and that their $\sin\alpha$ is our k.)

```
FUNCTION ellf(phi,ak)
REAL ellf,ak,phi
C  USES rf
```

Legendre elliptic integral of the 1st kind $F(\phi,k)$, evaluated using Carlson's function R_F. The argument ranges are $0\le\phi\le\pi/2,\ 0\le k\sin\phi\le 1$.

```
REAL s,rf
s=sin(phi)
ellf=s*rf(cos(phi)**2,(1.-s*ak)*(1.+s*ak),1.)
return
END
```

```
      FUNCTION elle(phi,ak)
      REAL elle,ak,phi
C     USES rd,rf
```

Legendre elliptic integral of the 2nd kind $E(\phi,k)$, evaluated using Carlson's functions R_D and R_F. The argument ranges are $0 \le \phi \le \pi/2$, $0 \le k\sin\phi \le 1$.

```
      REAL cc,q,s,rd,rf
      s=sin(phi)
      cc=cos(phi)**2
      q=(1.-s*ak)*(1.+s*ak)
      elle=s*(rf(cc,q,1.)-((s*ak)**2)*rd(cc,q,1.)/3.)
      return
      END

      FUNCTION ellpi(phi,en,ak)
      REAL ellpi,ak,en,phi
C     USES rf,rj
```

Legendre elliptic integral of the 3rd kind $\Pi(\phi,n,k)$, evaluated using Carlson's functions R_J and R_F. (Note that the sign convention on n is opposite that of Abramowitz and Stegun.) The ranges of ϕ and k are $0 \le \phi \le \pi/2$, $0 \le k\sin\phi \le 1$.

```
      REAL cc,enss,q,s,rf,rj
      s=sin(phi)
      enss=en*s*s
      cc=cos(phi)**2
      q=(1.-s*ak)*(1.+s*ak)
      ellpi=s*(rf(cc,q,1.)-enss*rj(cc,q,1.,1.+enss)/3.)
      return
      END
```

Carlson's functions are homogeneous of degree $-\frac{1}{2}$ and $-\frac{3}{2}$, so

$$\begin{aligned} R_F(\lambda x,\lambda y,\lambda z) &= \lambda^{-1/2}R_F(x,y,z) \\ R_J(\lambda x,\lambda y,\lambda z,\lambda p) &= \lambda^{-3/2}R_J(x,y,z,p) \end{aligned} \tag{6.11.22}$$

Thus to express a Carlson function in Legendre's notation, permute the first three arguments into ascending order, use homogeneity to scale the third argument to be 1, and then use equations (6.11.19)–(6.11.21).

Jacobian Elliptic Functions

The Jacobian elliptic function sn is defined as follows: instead of considering the elliptic integral

$$u(y,k) \equiv u = F(\phi,k) \tag{6.11.23}$$

consider the *inverse* function

$$y = \sin\phi = \mathrm{sn}(u,k) \tag{6.11.24}$$

Equivalently,

$$u = \int_0^{\mathrm{sn}} \frac{dy}{\sqrt{(1-y^2)(1-k^2y^2)}} \tag{6.11.25}$$

When $k = 0$, sn is just sin. The functions cn and dn are defined by the relations

$$\mathrm{sn}^2 + \mathrm{cn}^2 = 1, \qquad k^2\mathrm{sn}^2 + \mathrm{dn}^2 = 1 \tag{6.11.26}$$

The routine given below actually takes $m_c \equiv k_c^2 = 1 - k^2$ as an input parameter. It also computes all three functions sn, cn, and dn since computing all three is no harder than computing any one of them. For a description of the method, see [8].

```
      SUBROUTINE sncndn(uu,emmc,sn,cn,dn)
      REAL cn,dn,emmc,sn,uu,CA
      PARAMETER (CA=.0003)          The accuracy is the square of CA.
```

Returns the Jacobian elliptic functions sn(u, k_c), cn(u, k_c), and dn(u, k_c). Here uu $= u$, while emmc $= k_c^2$.

```
      INTEGER i,ii,l
      REAL a,b,c,d,emc,u,em(13),en(13)
      LOGICAL bo
      emc=emmc
      u=uu
      if(emc.ne.0.)then
        bo=(emc.lt.0.)
        if(bo)then
          d=1.-emc
          emc=-emc/d
          d=sqrt(d)
          u=d*u
        endif
        a=1.
        dn=1.
        do 11 i=1,13
          l=i
          em(i)=a
          emc=sqrt(emc)
          en(i)=emc
          c=0.5*(a+emc)
          if(abs(a-emc).le.CA*a)goto 1
          emc=a*emc
          a=c
        enddo 11
1       u=c*u
        sn=sin(u)
        cn=cos(u)
        if(sn.eq.0.)goto 2
        a=cn/sn
        c=a*c
        do 12 ii=l,1,-1
          b=em(ii)
          a=c*a
          c=dn*c
          dn=(en(ii)+a)/(b+a)
          a=c/b
        enddo 12
        a=1./sqrt(c**2+1.)
        if(sn.lt.0.)then
          sn=-a
        else
          sn=a
        endif
        cn=c*sn
2       if(bo)then
          a=dn
          dn=cn
          cn=a
          sn=sn/d
        endif
      else
        cn=1./cosh(u)
        dn=cn
        sn=tanh(u)
      endif
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendental Functions*, Vol. II, (New York: McGraw-Hill). [1]

Gradshteyn, I.S., and Ryzhik, I.W. 1980, *Table of Integrals, Series, and Products* (New York: Academic Press). [2]

Carlson, B.C. 1977, *SIAM Journal on Mathematical Analysis*, vol. 8, pp. 231–242. [3]

Carlson, B.C. 1987, *Mathematics of Computation*, vol. 49, pp. 595–606 [4]; 1988, *op. cit.*, vol. 51, pp. 267–280 [5]; 1989, *op. cit.*, vol. 53, pp. 327–333 [6]; 1991, *op. cit.*, vol. 56, pp. 267–280. [7]

Bulirsch, R. 1965, *Numerische Mathematik*, vol. 7, pp. 78–90; 1965, *op. cit.*, vol. 7, pp. 353–354; 1969, *op. cit.*, vol. 13, pp. 305–315. [8]

Carlson, B.C. 1979, *Numerische Mathematik*, vol. 33, pp. 1–16. [9]

Carlson, B.C., and Notis, E.M. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 398–403. [10]

Carlson, B.C. 1978, *SIAM Journal on Mathematical Analysis*, vol. 9, p. 524–528. [11]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 17. [12]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 78–79.

6.12 Hypergeometric Functions

As was discussed in §5.14, a fast, general routine for the the complex hypergeometric function ${}_2F_1(a,b,c;z)$, is difficult or impossible. The function is defined as the analytic continuation of the hypergeometric series,

$$
\begin{aligned}
{}_2F_1(a,b,c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots \\
+ \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots
\end{aligned}
\tag{6.12.1}
$$

This series converges only within the unit circle $|z| < 1$ (see [1]), but one's interest in the function is not confined to this region.

Section 5.14 discussed the method of evaluating this function by direct path integration in the complex plane. We here merely list the routines that result.

Implementation of the function `hypgeo` is straightforward, and is described by comments in the program. The machinery associated with Chapter 16's routine for integrating differential equations, `odeint`, is only minimally intrusive, and need not even be completely understood: use of `odeint` requires a common block with one zeroed variable, one subroutine call, and a prescribed format for the derivative routine `hypdrv`.

The subroutine `hypgeo` will fail, of course, for values of z too close to the singularity at 1. (If you need to approach this singularity, or the one at ∞, use the "linear transformation formulas" in §15.3 of [1].) Away from $z = 1$, and for moderate values of a, b, c, it is often remarkable how few steps are required to integrate the equations. A half-dozen is typical.

CITED REFERENCES AND FURTHER READING:

Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendental Functions*, Vol. II, (New York: McGraw-Hill). [1]

Gradshteyn, I.S., and Ryzhik, I.W. 1980, *Table of Integrals, Series, and Products* (New York: Academic Press). [2]

Carlson, B.C. 1977, *SIAM Journal on Mathematical Analysis*, vol. 8, pp. 231–242. [3]

Carlson, B.C. 1987, *Mathematics of Computation*, vol. 49, pp. 595–606 [4]; 1988, *op. cit.*, vol. 51, pp. 267–280 [5]; 1989, *op. cit.*, vol. 53, pp. 327–333 [6]; 1991, *op. cit.*, vol. 56, pp. 267–280. [7]

Bulirsch, R. 1965, *Numerische Mathematik*, vol. 7, pp. 78–90; 1965, *op. cit.*, vol. 7, pp. 353–354; 1969, *op. cit.*, vol. 13, pp. 305–315. [8]

Carlson, B.C. 1979, *Numerische Mathematik*, vol. 33, pp. 1–16. [9]

Carlson, B.C., and Notis, E.M. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 398–403. [10]

Carlson, B.C. 1978, *SIAM Journal on Mathematical Analysis*, vol. 9, p. 524–528. [11]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 17. [12]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 78–79.

6.12 Hypergeometric Functions

As was discussed in §5.14, a fast, general routine for the the complex hypergeometric function ${}_2F_1(a,b,c;z)$, is difficult or impossible. The function is defined as the analytic continuation of the hypergeometric series,

$$
\begin{aligned}
{}_2F_1(a,b,c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots \\
+ \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots
\end{aligned}
\tag{6.12.1}
$$

This series converges only within the unit circle $|z| < 1$ (see [1]), but one's interest in the function is not confined to this region.

Section 5.14 discussed the method of evaluating this function by direct path integration in the complex plane. We here merely list the routines that result.

Implementation of the function `hypgeo` is straightforward, and is described by comments in the program. The machinery associated with Chapter 16's routine for integrating differential equations, `odeint`, is only minimally intrusive, and need not even be completely understood: use of `odeint` requires a common block with one zeroed variable, one subroutine call, and a prescribed format for the derivative routine `hypdrv`.

The subroutine `hypgeo` will fail, of course, for values of z too close to the singularity at 1. (If you need to approach this singularity, or the one at ∞, use the "linear transformation formulas" in §15.3 of [1].) Away from $z = 1$, and for moderate values of a, b, c, it is often remarkable how few steps are required to integrate the equations. A half-dozen is typical.

```
      FUNCTION hypgeo(a,b,c,z)
      COMPLEX hypgeo,a,b,c,z
      REAL EPS
      PARAMETER (EPS=1.e-6)                    Accuracy parameter.
C     USES bsstep,hypdrv,hypser,odeint
```

Complex hypergeometric function ${}_2F_1$ for complex a, b, c, and z, by direct integration of the hypergeometric equation in the complex plane. The branch cut is taken to lie along the real axis, Re $z > 1$.

```
      INTEGER kmax,nbad,nok
      EXTERNAL bsstep,hypdrv
      COMPLEX z0,dz,aa,bb,cc,y(2)
      COMMON /hypg/ aa,bb,cc,z0,dz
      COMMON /path/ kmax                       Used by odeint.
      kmax=0
      if (real(z)**2+aimag(z)**2.le.0.25) then Use series...
          call hypser(a,b,c,z,hypgeo,y(2))
          return
      else if (real(z).lt.0.) then             ...or pick a starting point for the path inte-
          z0=cmplx(-0.5,0.)                        gration.
      else if (real(z).le.1.0) then
          z0=cmplx(0.5,0.)
      else
          z0=cmplx(0.,sign(0.5,aimag(z)))
      endif
      aa=a                                     Load the common block, used to pass pa-
      bb=b                                         rameters "over the head" of odeint to
      cc=c                                         hypdrv.
      dz=z-z0
      call hypser(aa,bb,cc,z0,y(1),y(2))       Get starting function and derivative.
      call odeint(y,4,0.,1.,EPS,.1,.0001,nok,nbad,hypdrv,bsstep)
```

The arguments to odeint are the vector of independent variables, its length, the starting and ending values of the dependent variable, the accuracy parameter, an initial guess for stepsize, a minimum stepsize, the (returned) number of good and bad steps taken, and the names of the derivative routine and the (here Bulirsch-Stoer) stepping routine.

```
      hypgeo=y(1)
      return
      END
```

```
      SUBROUTINE hypser(a,b,c,z,series,deriv)
      INTEGER n
      COMPLEX a,b,c,z,series,deriv,aa,bb,cc,fac,temp
```

Returns the hypergeometric series ${}_2F_1$ and its derivative, iterating to machine accuracy. For cabs(z) $\leq 1/2$ convergence is quite rapid.

```
      deriv=cmplx(0.,0.)
      fac=cmplx(1.,0.)
      temp=fac
      aa=a
      bb=b
      cc=c
      do 11 n=1,1000
          fac=((aa*bb)/cc)*fac
          deriv=deriv+fac
          fac=fac*z/n
          series=temp+fac
          if (series.eq.temp) return
          temp=series
          aa=aa+1.
          bb=bb+1.
          cc=cc+1.
      enddo 11
      pause 'convergence failure in hypser'
      END
```

```
SUBROUTINE hypdrv(s,y,dyds)
REAL s
COMPLEX y(2),dyds(2),aa,bb,cc,z0,dz,z
    Derivative subroutine for the hypergeometric equation, see text equation (5.14.4).
COMMON /hypg/ aa,bb,cc,z0,dz
z=z0+s*dz
dyds(1)=y(2)*dz
dyds(2)=((aa*bb)*y(1)-(cc-((aa+bb)+1.)*z)*y(2))*dz/(z*(1.-z))
return
END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [1]

Chapter 7. Random Numbers

7.0 Introduction

It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce "random" numbers. More than perverse, it may seem to be a conceptual impossibility. Any program, after all, will produce output that is entirely predictable, hence not truly "random."

Nevertheless, practical computer "random number generators" are in common use. We will leave it to philosophers of the computer age to resolve the paradox in a deep way (see, e.g., Knuth [1] §3.5 for discussion and references). One sometimes hears computer-generated sequences termed *pseudo-random*, while the word *random* is reserved for the output of an intrinsically random physical process, like the elapsed time between clicks of a Geiger counter placed next to a sample of some radioactive element. We will not try to make such fine distinctions.

A working, though imprecise, definition of randomness in the context of computer-generated sequences, is to say that the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that *uses* its output. In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular applications program. If they don't, then at least one of them is not (from your point of view) a good generator.

The above definition may seem circular, comparing, as it does, one generator to another. However, there exists a body of random number generators which mutually do satisfy the definition over a very, very broad class of applications programs. And it is also found empirically that statistically identical results are obtained from random numbers produced by physical processes. So, because such generators are known to exist, we can leave to the philosophers the problem of defining them.

A pragmatic point of view, then, is that randomness is in the eye of the beholder (or programmer). What is random enough for one application may not be random enough for another. Still, one is not entirely adrift in a sea of incommensurable applications programs: There is a certain list of statistical tests, some sensible and some merely enshrined by history, which on the whole will do a very good job of ferreting out any correlations that are likely to be detected by an applications program (in this case, yours). Good random number generators ought to pass all of these tests; or at least the user had better be aware of any that they fail, so that he or she will be able to judge whether they are relevant to the case at hand.

As for references on this subject, the one to turn to first is Knuth [1]. Then try [2]. Only a few of the standard books on numerical methods [3-4] treat topics relating to random numbers.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5. [1]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [2]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 11. [3]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10. [4]

7.1 Uniform Deviates

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think "random numbers" are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

System-Supplied Random Number Generators

Your computer very likely has lurking within it a library routine which is called a "random number generator." That routine typically has an unforgettable name like "`ran`," and a calling sequence like

```
x=ran(iseed)          sets x to the next random number and updates iseed
```

You initialize `iseed` to a (usually) arbitrary value before the first call to `ran`. Each initializing value will typically return a different subsequent random sequence, or at least a different subsequence of some one enormously long sequence. The *same* initializing value of `iseed` will always return the *same* random sequence, however.

Now our first, and perhaps most important, lesson in this chapter is: Be *very, very* suspicious of a system-supplied `ran` that resembles the one just described. If all scientific papers whose results are in doubt because of bad `ran`s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist. System-supplied `ran`s are almost always *linear congruential generators*, which

As for references on this subject, the one to turn to first is Knuth [1]. Then try [2]. Only a few of the standard books on numerical methods [3-4] treat topics relating to random numbers.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5. [1]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [2]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 11. [3]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10. [4]

7.1 Uniform Deviates

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think "random numbers" are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

System-Supplied Random Number Generators

Your computer very likely has lurking within it a library routine which is called a "random number generator." That routine typically has an unforgettable name like "`ran`," and a calling sequence like

```
x=ran(iseed)          sets x to the next random number and updates iseed
```

You initialize `iseed` to a (usually) arbitrary value before the first call to `ran`. Each initializing value will typically return a different subsequent random sequence, or at least a different subsequence of some one enormously long sequence. The *same* initializing value of `iseed` will always return the *same* random sequence, however.

Now our first, and perhaps most important, lesson in this chapter is: Be *very, very* suspicious of a system-supplied `ran` that resembles the one just described. If all scientific papers whose results are in doubt because of bad `ran`s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist. System-supplied `ran`s are almost always *linear congruential generators*, which

generate a sequence of integers $I_1, I_2, I_3, \ldots$, each between 0 and $m-1$ (a large number) by the recurrence relation

$$I_{j+1} = aI_j + c \pmod{m} \tag{7.1.1}$$

Here m is called the *modulus*, and a and c are positive integers called the *multiplier* and the *increment*, respectively. The recurrence (7.1.1) will eventually repeat itself, with a period that is obviously no greater than m. If m, a, and c are properly chosen, then the period will be of maximal length, i.e., of length m. In that case, all possible integers between 0 and $m-1$ occur at some point, so any initial "seed" choice of I_0 is as good as any other: The sequence just takes off from that point. The real number between 0 and 1 which is returned is generally I_{j+1}/m, so that it is strictly less than 1, but occasionally (once in m calls) exactly equal to zero. `iseed` is set to I_{j+1} (or some encoding of it), so that it can be used on the next call to generate I_{j+2}, and so on.

The linear congruential method has the advantage of being very fast, requiring only a few operations per call, hence its almost universal use. It has the disadvantage that it is not free of sequential correlation on successive calls. If k random numbers at a time are used to plot points in k dimensional space (with each coordinate between 0 and 1), then the points will not tend to "fill up" the k-dimensional space, but rather will lie on $(k-1)$-dimensional "planes." There will be *at most* about $m^{1/k}$ such planes. If the constants m, a, and c are not very carefully chosen, there will be *many fewer than that.* The number m is usually close to the machine's largest representable integer, e.g., $\sim 2^{32}$. So, for example, the number of planes on which triples of points lie in three-dimensional space is usually no greater than about the cube root of 2^{32}, about 1600. You might well be focusing attention on a physical process that occurs in a small fraction of the total volume, so that the discreteness of the planes can be very pronounced.

Even worse, you might be using a `ran` whose choices of m, a, and c have been botched. One infamous such routine, `RANDU`, with $a = 65539$ and $m = 2^{31}$, was widespread on IBM mainframe computers for many years, and widely copied onto other systems [1]. One of us recalls producing a "random" plot with only 11 planes, and being told by his computer center's programming consultant that he had misused the random number generator: "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random." Figure that out.

Correlation in k-space is not the only weakness of linear congruential generators. Such generators often have their low-order (least significant) bits much less random than their high-order bits. If you want to generate a random integer between 1 and 10, you should always do it using high-order bits, as in

```
j=1+int(10.*ran(iseed))
```

and never by anything resembling

```
j=1+mod(int(1000000.*ran(iseed)),10)
```

(which uses lower-order bits). Similarly you should never try to take apart a "`ran`" number into several supposedly random pieces. Instead use separate calls for every piece.

Portable Random Number Generators

Park and Miller [1] have surveyed a large number of random number generators that have been used over the last 30 years or more. Along with a good theoretical review, they present an anecdotal sampling of a number of inadequate generators that have come into widespread use. The historical record is nothing if not appalling.

There is good evidence, both theoretical and empirical, that the simple multiplicative congruential algorithm

$$I_{j+1} = aI_j \pmod m \tag{7.1.2}$$

can be as good as any of the more general linear congruential generators that have $c \neq 0$ (equation 7.1.1) — *if* the multiplier a and modulus m are chosen exquisitely carefully. Park and Miller propose a "Minimal Standard" generator based on the choices

$$a = 7^5 = 16807 \qquad m = 2^{31} - 1 = 2147483647 \tag{7.1.3}$$

First proposed by Lewis, Goodman, and Miller in 1969, this generator has in subsequent years passed all new theoretical tests, and (perhaps more importantly) has accumulated a large amount of successful use. Park and Miller do not claim that the generator is "perfect" (we will see below that it is not), but only that it is a good minimal standard against which other generators should be judged.

It is not possible to implement equations (7.1.2) and (7.1.3) directly in a high-level language, since the product of a and $m - 1$ exceeds the maximum value for a 32-bit integer. Assembly language implementation using a 64-bit product register is straightforward, but not portable from machine to machine. A trick due to Schrage [2,3] for multiplying two 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits (including a sign bit) is therefore extremely interesting: It allows the Minimal Standard generator to be implemented in essentially any programming language on essentially any machine.

Schrage's algorithm is based on an *approximate factorization* of m,

$$m = aq + r, \quad \text{i.e.,} \quad q = [m/a],\ r = m \bmod a \tag{7.1.4}$$

with square brackets denoting integer part. If r is small, specifically $r < q$, and $0 < z < m - 1$, it can be shown that both $a(z \bmod q)$ and $r[z/q]$ lie in the range $0, \ldots, m - 1$, and that

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{if it is} \geq 0, \\ a(z \bmod q) - r[z/q] + m & \text{otherwise} \end{cases} \tag{7.1.5}$$

The application of Schrage's algorithm to the constants (7.1.3) uses the values $q = 127773$ and $r = 2836$.

Here is an implementation of the Minimal Standard generator:

```
      FUNCTION ran0(idum)
      INTEGER idum,IA,IM,IQ,IR,MASK
      REAL ran0,AM
      PARAMETER (IA=16807,IM=2147483647,AM=1./IM,
     *     IQ=127773,IR=2836,MASK=123459876)
        "Minimal" random number generator of Park and Miller. Returns a uniform random deviate
        between 0.0 and 1.0. Set or reset idum to any integer value (except the unlikely value MASK)
        to initialize the sequence; idum must not be altered between calls for successive deviates
        in a sequence.
      INTEGER k
      idum=ieor(idum,MASK)              XORing with MASK allows use of zero and other simple
      k=idum/IQ                             bit patterns for idum.
      idum=IA*(idum-k*IQ)-IR*k          Compute idum=mod(IA*idum,IM) without overflows by
      if (idum.lt.0) idum=idum+IM           Schrage's method.
      ran0=AM*idum                      Convert idum to a floating result.
      idum=ieor(idum,MASK)              Unmask before return.
      return
      END
```

The period of `ran0` is $2^{31} - 2 \approx 2.1 \times 10^9$. A peculiarity of generators of the form (7.1.2) is that the value 0 must never be allowed as the initial seed — it perpetuates itself — and it never occurs for any nonzero initial seed. Experience has shown that users always manage to call random number generators with the seed `idum=0`. That is why `ran0` performs its exclusive-or with an arbitrary constant both on entry and exit. If you are the first user in history to be proof against human error, you can remove the two lines with the `ieor` function.

Park and Miller discuss two other multipliers a that can be used with the same $m = 2^{31} - 1$. These are $a = 48271$ (with $q = 44488$ and $r = 3399$) and $a = 69621$ (with $q = 30845$ and $r = 23902$). These can be substituted in the routine `ran0` if desired; they may be slightly superior to Lewis *et al.*'s longer-tested values. No values other than these should be used.

The routine `ran0` is a Minimal Standard, satisfactory for the majority of applications, but we do not recommend it as the final word on random number generators. Our reason is precisely the simplicity of the Minimal Standard. It is not hard to think of situations where successive random numbers might be used in a way that accidentally conflicts with the generation algorithm. For example, since successive numbers differ by a multiple of only 1.6×10^4 out of a modulus of more than 2×10^9, very small random numbers will tend to be followed by smaller than average values. One time in 10^6, for example, there will be a value $< 10^{-6}$ returned (as there should be), but this will *always* be followed by a value less than about 0.0168. One can easily think of applications involving rare events where this property would lead to wrong results.

There are other, more subtle, serial correlations present in `ran0`. For example, if successive points (I_i, I_{i+1}) are binned into a two-dimensional plane for $i = 1, 2, \ldots, N$, then the resulting distribution fails the χ^2 test when N is greater than a few $\times 10^7$, much less than the period $m - 2$. Since low-order serial correlations have historically been such a bugaboo, and since there is a very simple way to remove them, we think that it is prudent to do so.

The following routine, `ran1`, uses the Minimal Standard for its random value, but it shuffles the output to remove low-order serial correlations. A random deviate derived from the jth value in the sequence, I_j, is output not on the jth call, but rather on a randomized later call, $j + 32$ on average. The shuffling algorithm is due to Bays and Durham as described in Knuth [4], and is illustrated in Figure 7.1.1.

```
      FUNCTION ran1(idum)
      INTEGER idum,IA,IM,IQ,IR,NTAB,NDIV
      REAL ran1,AM,EPS,RNMX
      PARAMETER (IA=16807,IM=2147483647,AM=1./IM,IQ=127773,IR=2836,
     *     NTAB=32,NDIV=1+(IM-1)/NTAB,EPS=1.2e-7,RNMX=1.-EPS)
```

"Minimal" random number generator of Park and Miller with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with `idum` a negative integer to initialize; thereafter, do not alter `idum` between successive deviates in a sequence. `RNMX` should approximate the largest floating value that is less than 1.

```
      INTEGER j,k,iv(NTAB),iy
      SAVE iv,iy
      DATA iv /NTAB*0/, iy /0/
      if (idum.le.0.or.iy.eq.0) then  Initialize.
        idum=max(-idum,1)             Be sure to prevent idum = 0.
        do 11 j=NTAB+8,1,-1           Load the shuffle table (after 8 warm-ups).
          k=idum/IQ
          idum=IA*(idum-k*IQ)-IR*k
          if (idum.lt.0) idum=idum+IM
          if (j.le.NTAB) iv(j)=idum
        enddo 11
        iy=iv(1)
      endif
      k=idum/IQ                       Start here when not initializing.
      idum=IA*(idum-k*IQ)-IR*k        Compute idum=mod(IA*idum,IM) without overflows by
      if (idum.lt.0) idum=idum+IM       Schrage's method.
      j=1+iy/NDIV                     Will be in the range 1:NTAB.
      iy=iv(j)                        Output previously stored value and refill the shuffle ta-
      iv(j)=idum                        ble.
      ran1=min(AM*iy,RNMX)            Because users don't expect endpoint values.
      return
      END
```

The routine `ran1` passes those statistical tests that `ran0` is known to fail. In fact, we do not know of any statistical test that `ran1` fails to pass, except when the number of calls starts to become on the order of the period m, say $> 10^8 \approx m/20$.

For situations when even longer random sequences are needed, L'Ecuyer [6] has given a good way of combining two different sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods. The basic idea is simply to add the two sequences, modulo the modulus of *either* of them (call it m). A trick to avoid an intermediate value that overflows the integer wordsize is to subtract rather than add, and then add back the constant $m - 1$ if the result is ≤ 0, so as to wrap around into the desired interval $0, \ldots, m - 1$.

Notice that it is not necessary that this wrapped subtraction be able to reach all values $0, \ldots, m - 1$ from *every* value of the first sequence. Consider the absurd extreme case where the value subtracted was only between 1 and 10: The resulting sequence would still be no less random than the first sequence by itself. As a practical matter it is only necessary that the second sequence have a range covering *substantially* all of the range of the first. L'Ecuyer recommends the use of the two generators $m_1 = 2147483563$ (with $a_1 = 40014$, $q_1 = 53668$, $r_1 = 12211$) and $m_2 = 2147483399$ (with $a_2 = 40692$, $q_2 = 52774$, $r_2 = 3791$). Both moduli are slightly less than 2^{31}. The periods $m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031$ and $m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$ share only the factor 2, so the period of the combined generator is $\approx 2.3 \times 10^{18}$. For present computers, period exhaustion is a practical impossibility.

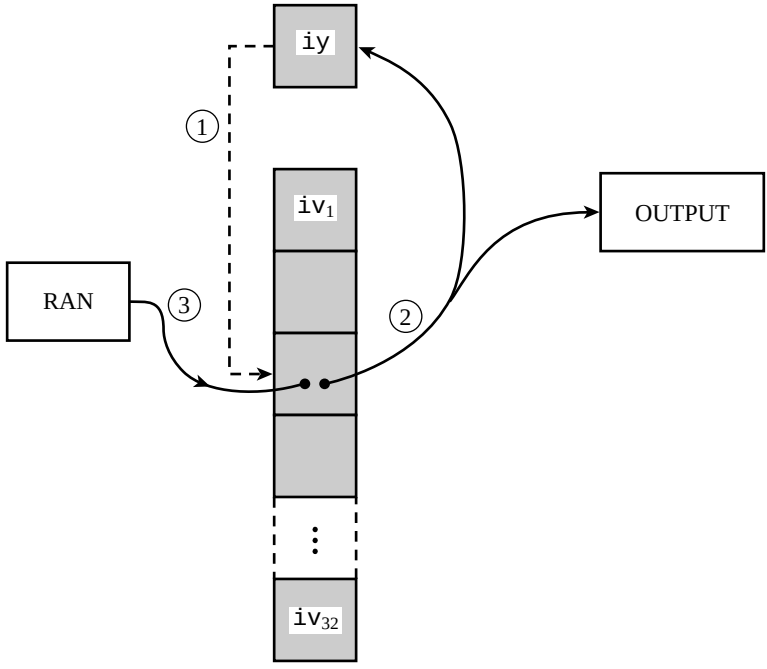


Figure 7.1.1. Shuffling procedure used in ran1 to break up sequential correlations in the Minimal Standard generator. Circled numbers indicate the sequence of events: On each call, the random number in iy is used to choose a random element in the array iv. That element becomes the output random number, and also is the next iy. Its spot in iv is refilled from the Minimal Standard routine.

Combining the two generators breaks up serial correlations to a considerable extent. We nevertheless recommend the additional shuffle that is implemented in the following routine, ran2. We think that, within the limits of its floating-point precision, ran2 provides perfect random numbers; a practical definition of "perfect" is that we will pay $1000 to the first reader who convinces us otherwise (by finding a statistical test that ran2 fails in a nontrivial way, excluding the ordinary limitations of a machine's floating-point representation).

```
      FUNCTION ran2(idum)
      INTEGER idum,IM1,IM2,IMM1,IA1,IA2,IQ1,IQ2,IR1,IR2,NTAB,NDIV
      REAL ran2,AM,EPS,RNMX
      PARAMETER (IM1=2147483563,IM2=2147483399,AM=1./IM1,IMM1=IM1-1,
     *     IA1=40014,IA2=40692,IQ1=53668,IQ2=52774,IR1=12211,
     *     IR2=3791,NTAB=32,NDIV=1+IMM1/NTAB,EPS=1.2e-7,RNMX=1.-EPS)
        Long period (> 2 × 10^18) random number generator of L'Ecuyer with Bays-Durham shuffle
        and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive
        of the endpoint values). Call with idum a negative integer to initialize; thereafter, do not
        alter idum between successive deviates in a sequence. RNMX should approximate the largest
        floating value that is less than 1.
      INTEGER idum2,j,k,iv(NTAB),iy
      SAVE iv,iy,idum2
      DATA idum2/123456789/, iv/NTAB*0/, iy/0/
      if (idum.le.0) then                    Initialize.
        idum=max(-idum,1)                    Be sure to prevent idum = 0.
        idum2=idum
        do 11 j=NTAB+8,1,-1                  Load the shuffle table (after 8 warm-ups).
          k=idum/IQ1
```

```
        idum=IA1*(idum-k*IQ1)-k*IR1
        if (idum.lt.0) idum=idum+IM1
        if (j.le.NTAB) iv(j)=idum
      enddo 11
      iy=iv(1)
    endif
    k=idum/IQ1                          Start here when not initializing.
    idum=IA1*(idum-k*IQ1)-k*IR1         Compute idum=mod(IA1*idum,IM1) without over-
    if (idum.lt.0) idum=idum+IM1            flows by Schrage's method.
    k=idum2/IQ2
    idum2=IA2*(idum2-k*IQ2)-k*IR2       Compute idum2=mod(IA2*idum2,IM2) likewise.
    if (idum2.lt.0) idum2=idum2+IM2
    j=1+iy/NDIV                         Will be in the range 1:NTAB.
    iy=iv(j)-idum2                      Here idum is shuffled, idum and idum2 are com-
    iv(j)=idum                              bined to generate output.
    if(iy.lt.1)iy=iy+IMM1
    ran2=min(AM*iy,RNMX)                Because users don't expect endpoint values.
    return
    END
```

L'Ecuyer [6] lists additional short generators that can be combined into longer ones, including generators that can be implemented in 16-bit integer arithmetic.

Finally, we give you Knuth's suggestion [4] for a portable routine, which we have translated to the present conventions as `ran3`. This is not based on the linear congruential method at all, but rather on a *subtractive method* (see also [5]). One might hope that its weaknesses, if any, are therefore of a highly different character from the weaknesses, if any, of `ran1` above. If you ever suspect trouble with one routine, it is a good idea to try the other in the same application. `ran3` has one nice feature: if your machine is poor on integer arithmetic (i.e., is limited to 16-bit integers), substitution of the three "commented" lines for the ones directly preceding them will render the routine entirely floating-point.

```
    FUNCTION ran3(idum)
      Returns a uniform random deviate between 0.0 and 1.0. Set idum to any negative value
      to initialize or reinitialize the sequence.
    INTEGER idum
    INTEGER MBIG,MSEED,MZ
C   REAL MBIG,MSEED,MZ
    REAL ran3,FAC
    PARAMETER (MBIG=1000000000,MSEED=161803398,MZ=0,FAC=1./MBIG)
C   PARAMETER (MBIG=4000000.,MSEED=1618033.,MZ=0.,FAC=1./MBIG)
      According to Knuth, any large mbig, and any smaller (but still large) mseed can be sub-
      stituted for the above values.
    INTEGER i,iff,ii,inext,inextp,k
    INTEGER mj,mk,ma(55)                The value 55 is special and should not be modified; see
C   REAL mj,mk,ma(55)                       Knuth.
    SAVE iff,inext,inextp,ma
    DATA iff /0/
    if(idum.lt.0.or.iff.eq.0)then       Initialization.
      iff=1
      mj=abs(MSEED-abs(idum))           Initialize ma(55) using the seed idum and the large num-
      mj=mod(mj,MBIG)                       ber mseed.
      ma(55)=mj
      mk=1
      do 11 i=1,54                      Now initialize the rest of the table,
        ii=mod(21*i,55)                 in a slightly random order,
        ma(ii)=mk                       with numbers that are not especially random.
        mk=mj-mk
        if(mk.lt.MZ)mk=mk+MBIG
```

```
        mj=ma(ii)
    enddo 11
    do 13 k=1,4                    We randomize them by "warming up the generator."
        do 12 i=1,55
            ma(i)=ma(i)-ma(1+mod(i+30,55))
            if(ma(i).lt.MZ)ma(i)=ma(i)+MBIG
        enddo 12
    enddo 13
    inext=0                        Prepare indices for our first generated number.
    inextp=31                      The constant 31 is special; see Knuth.
    idum=1
endif
inext=inext+1                      Here is where we start, except on initialization. Increment
if(inext.eq.56)inext=1                 inext, wrapping around 56 to 1.
inextp=inextp+1                    Ditto for inextp.
if(inextp.eq.56)inextp=1
mj=ma(inext)-ma(inextp)            Now generate a new random number subtractively.
if(mj.lt.MZ)mj=mj+MBIG             Be sure that it is in range.
ma(inext)=mj                       Store it,
ran3=mj*FAC                        and output the derived uniform deviate.
return
END
```

Quick and Dirty Generators

One sometimes would like a "quick and dirty" generator to embed in a program, perhaps taking only one or two lines of code, just to *somewhat* randomize things. One might wish to process data from an experiment not always in exactly the same order, for example, so that the first output is more "typical" than might otherwise be the case.

For this kind of application, all we really need is a list of "good" choices for m, a, and c in equation (7.1.1). If we don't need a period longer than 10^4 to 10^6, say, we can keep the value of $(m-1)a+c$ small enough to avoid overflows that would otherwise mandate the extra complexity of Schrage's method (above). We can thus easily embed in our programs

```
jran=mod(jran*ia+ic,im)
ran=float(jran)/float(im)
```

whenever we want a quick and dirty uniform deviate, or

```
jran=mod(jran*ia+ic,im)
j=jlo+((jhi-jlo+1)*jran)/im
```

whenever we want an integer between `jlo` and `jhi`, inclusive. (In both cases `jran` was once initialized to any seed value between 0 and `im-1`.)

Be sure to remember, however, that when `im` is small, the kth root of it, which is the number of planes in k-space, is even smaller! So a quick and dirty generator should never be used to select points in k-space with $k > 1$.

With these caveats, some "good" choices for the constants are given in the accompanying table. These constants (i) give a period of maximal length `im`, and, more important, (ii) pass Knuth's "spectral test" for dimensions 2, 3, 4, 5, and 6. The increment `ic` is a prime, close to the value $(\frac{1}{2}-\frac{1}{6}\sqrt{3})$`im`; actually almost any value of `ic` that is relatively prime to `im` will do just as well, but there is some "lore" favoring this choice (see [4], p. 84).

Constants for Quick and Dirty Random Number Generators

overflow at	im	ia	ic	overflow at	im	ia	ic
	6075	106	1283		86436	1093	18257
2^{20}					121500	1021	25673
	7875	211	1663		259200	421	54773
2^{21}				2^{27}			
	7875	421	1663		117128	1277	24749
2^{22}					121500	2041	25673
	6075	1366	1283		312500	741	66037
	6655	936	1399	2^{28}			
	11979	430	2531		145800	3661	30809
2^{23}					175000	2661	36979
	14406	967	3041		233280	1861	49297
	29282	419	6173		244944	1597	51749
	53125	171	11213	2^{29}			
2^{24}					139968	3877	29573
	12960	1741	2731		214326	3613	45289
	14000	1541	2957		714025	1366	150889
	21870	1291	4621	2^{30}			
	31104	625	6571		134456	8121	28411
	139968	205	29573		259200	7141	54773
2^{25}				2^{31}			
	29282	1255	6173		233280	9301	49297
	81000	421	17117		714025	4096	150889
	134456	281	28411	2^{32}			
2^{26}							

An Even Quicker and Dirtier Generator

Many `FORTRAN` compilers can be abused in such a way that they will multiply two 32-bit integers *ignoring any resulting overflow*. In such cases, on many machines, the value returned is predictably the low-order 32 bits of the true 64-bit product. (`C` compilers, incidentally, can do this without the requirement of abuse — it is guaranteed behavior for so-called `unsigned long int` integers. On VMS VAXes, the necessary FORTRAN command is `FORTRAN/CHECK=NOOVERFLOW`.) If we now choose $m = 2^{32}$, the "mod" in equation (7.1.1) is free, and we have simply

$$I_{j+1} = aI_j + c \tag{7.1.6}$$

Knuth suggests $a = 1664525$ as a suitable multiplier for this value of m. H.W. Lewis has conducted extensive tests of this value of a with $c = 1013904223$, which is a prime close to $(\sqrt{5} - 2)m$. The resulting in-line generator (we will call it `ranqd1`) is simply

```
idum=1664525*idum+1013904223
```

This is about as good as any 32-bit linear congruential generator, entirely adequate for many uses. And, with only a single multiply and add, it is *very* fast.

To check whether your compiler and machine have the desired overflow properties, see if you can generate the following sequence of 32-bit values (given here in hex): 00000000, 3C6EF35F, 47502932, D1CCF6E9, AAF95334, 6252E503, 9F2EC686, 57FE6C2D, A3D95FA8, 81FDBEE7, 94F0AF1A, CBF633B1.

If you need floating-point values instead of 32-bit integers, and want to avoid a divide by floating-point 2^{32}, a dirty trick is to mask in an exponent that makes the value lie between 1 and 2, then subtract 1.0. The resulting in-line generator (call it `ranqd2`) will look something like

```
      INTEGER idum,itemp,jflone,jflmsk
      REAL ftemp
      EQUIVALENCE (itemp,ftemp)
      DATA jflone /Z'3F800000'/, jflmsk /Z'007FFFFF'/
C     ...
      idum=1664525*idum+1013904223
      itemp=ior(jflone,iand(jflmsk,idum))
      ran=ftemp-1.0
```

The hex constants 3F800000 and 007FFFFF are the appropriate ones for computers using the IEEE representation for 32-bit floating-point numbers (e.g., IBM PCs and most UNIX workstations). For DEC VAXes, the correct hex constants are, respectively, 00004080 and FFFF007F. Notice that the IEEE mask results in the floating-point number being constructed out of the 23 low-order bits of the integer, which is not ideal. Also notice that your compiler may require a different notation for hex constants, e.g., `x'3f800000'`, `'3F800000'X`, or even `16#3F800000`. (Your authors have tried very hard to make *almost all* of the material in this book machine and compiler independent — indeed, even programming language independent. This subsection is a rare aberration. Forgive us. Once in a great while the temptation to be *really dirty* is just irresistible.)

Relative Timings and Recommendations

Timings are inevitably machine dependent. Nevertheless the following table is indicative of the *relative* timings, for typical machines, of the various uniform generators discussed in this section, plus `ran4` from §7.5. Smaller values in the table indicate faster generators. The generators `ranqd1` and `ranqd2` refer to the "quick and dirty" generators immediately above.

Generator	Relative Execution Time
`ran0`	$\equiv 1.0$
`ran1`	≈ 1.3
`ran2`	≈ 2.0
`ran3`	≈ 0.6
`ranqd1`	≈ 0.10
`ranqd2`	≈ 0.25
`ran4`	≈ 4.0

On balance, we recommend `ran1` for general use. It is portable, based on Park and Miller's Minimal Standard generator with an additional shuffle, and has no known (to us) flaws other than period exhaustion.

If you are generating more than 100,000,000 random numbers in a single calculation (that is, more than about 5% of `ran1`'s period), we recommend the use of `ran2`, with its much longer period.

Knuth's subtractive routine `ran3` seems to be the timing winner among portable routines. Unfortunately the subtractive method is not so well studied, and not a standard. We like to keep `ran3` in reserve for a "second opinion," substituting it when we suspect another generator of introducing unwanted correlations into a calculation.

The routine `ran4` generates *extremely* good random deviates, and has some other nice properties, but it is slow. See §7.5 for discussion.

Finally, the quick and dirty in-line generators `ranqd1` and `ranqd2` are very fast, but they are machine dependent, nonportable, and at best only as good as a 32-bit linear congruential generator ever is — in our view not good enough in many situations. We would use these only in very special cases, where speed is critical.

CITED REFERENCES AND FURTHER READING:

Park, S.K., and Miller, K.W. 1988, *Communications of the ACM*, vol. 31, pp. 1192–1201. [1]

Schrage, L. 1979, *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §§3.2–3.3. [4]

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 10. [5]

L'Ecuyer, P. 1988, *Communications of the ACM*, vol. 31, pp. 742–774. [6]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

7.2 Transformation Method: Exponential and Normal Deviates

In the previous section, we learned how to generate random deviates with a uniform probability distribution, so that the probability of generating a number between x and $x + dx$, denoted $p(x)dx$, is given by

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.2.1}$$

The probability distribution $p(x)$ is of course normalized, so that

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

Now suppose that we generate a uniform deviate x and then take some prescribed function of it, $y(x)$. The probability distribution of y, denoted $p(y)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$

Finally, the quick and dirty in-line generators `ranqd1` and `ranqd2` are very fast, but they are machine dependent, nonportable, and at best only as good as a 32-bit linear congruential generator ever is — in our view not good enough in many situations. We would use these only in very special cases, where speed is critical.

CITED REFERENCES AND FURTHER READING:

Park, S.K., and Miller, K.W. 1988, *Communications of the ACM*, vol. 31, pp. 1192–1201. [1]

Schrage, L. 1979, *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §§3.2–3.3. [4]

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 10. [5]

L'Ecuyer, P. 1988, *Communications of the ACM*, vol. 31, pp. 742–774. [6]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

7.2 Transformation Method: Exponential and Normal Deviates

In the previous section, we learned how to generate random deviates with a uniform probability distribution, so that the probability of generating a number between x and $x + dx$, denoted $p(x)dx$, is given by

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.2.1}$$

The probability distribution $p(x)$ is of course normalized, so that

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

Now suppose that we generate a uniform deviate x and then take some prescribed function of it, $y(x)$. The probability distribution of y, denoted $p(y)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$

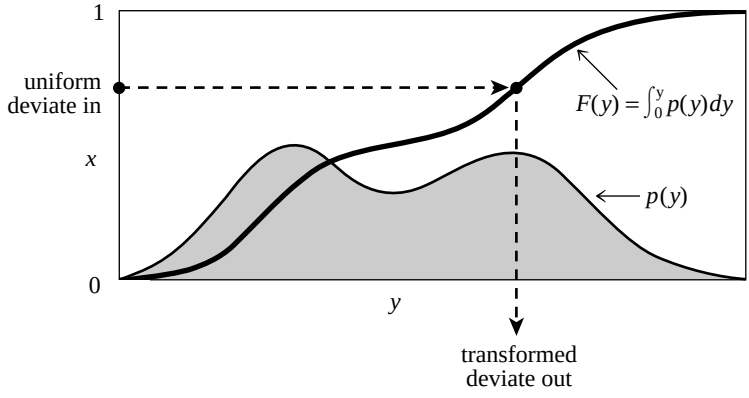


Figure 7.2.1. Transformation method for generating a random deviate y from a known probability distribution $p(y)$. The indefinite integral of $p(y)$ must be known and invertible. A uniform deviate x is chosen between 0 and 1. Its corresponding y on the definite-integral curve is the desired deviate.

Exponential Deviates

As an example, suppose that $y(x) \equiv -\ln(x)$, and that $p(x)$ is as given by equation (7.2.1) for a uniform deviate. Then

$$p(y)dy = \left|\frac{dx}{dy}\right| dy = e^{-y}dy \tag{7.2.5}$$

which is distributed exponentially. This exponential distribution occurs frequently in real problems, usually as the distribution of waiting times between independent Poisson-random events, for example the radioactive decay of nuclei. You can also easily see (from 7.2.4) that the quantity y/λ has the probability distribution $\lambda e^{-\lambda y}$.

So we have

```
      FUNCTION expdev(idum)
      INTEGER idum
      REAL expdev
C     USES ran1
        Returns an exponentially distributed, positive, random deviate of unit mean, using
        ran1(idum) as the source of uniform deviates.
      REAL dum,ran1
1     dum=ran1(idum)
      if(dum.eq.0.)goto 1
      expdev=-log(dum)
      return
      END
```

Let's see what is involved in using the above *transformation method* to generate some arbitrary desired distribution of y's, say one with $p(y) = f(y)$ for some positive function f whose integral is 1. (See Figure 7.2.1.) According to (7.2.4), we need to solve the differential equation

$$\frac{dx}{dy} = f(y) \tag{7.2.6}$$

But the solution of this is just $x = F(y)$, where $F(y)$ is the indefinite integral of $f(y)$. The desired transformation which takes a uniform deviate into one distributed as $f(y)$ is therefore

$$y(x) = F^{-1}(x) \tag{7.2.7}$$

where F^{-1} is the inverse function to F. Whether (7.2.7) is feasible to implement depends on whether the *inverse function of the integral of f(y)* is itself feasible to compute, either analytically or numerically. Sometimes it is, and sometimes it isn't.

Incidentally, (7.2.7) has an immediate geometric interpretation: Since $F(y)$ is the area under the probability curve to the left of y, (7.2.7) is just the prescription: choose a uniform random x, then find the value y that has that fraction x of probability area to its left, and return the value y.

Normal (Gaussian) Deviates

Transformation methods generalize to more than one dimension. If $x_1, x_2, \ldots$ are random deviates with a *joint* probability distribution $p(x_1, x_2, \ldots)\, dx_1 dx_2 \ldots$, and if $y_1, y_2, \ldots$ are each functions of all the x's (same number of y's as x's), then the joint probability distribution of the y's is

$$p(y_1, y_2, \ldots) dy_1 dy_2 \ldots = p(x_1, x_2, \ldots) \left| \frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)} \right| dy_1 dy_2 \ldots \tag{7.2.8}$$

where $|\partial(\quad)/\partial(\quad)|$ is the Jacobian determinant of the x's with respect to the y's (or reciprocal of the Jacobian determinant of the y's with respect to the x's).

An important example of the use of (7.2.8) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution,

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.2.9}$$

Consider the transformation between two uniform deviates on (0,1), x_1, x_2, and two quantities y_1, y_2,

$$\begin{aligned} y_1 &= \sqrt{-2 \ln x_1} \cos 2\pi x_2 \\ y_2 &= \sqrt{-2 \ln x_1} \sin 2\pi x_2 \end{aligned} \tag{7.2.10}$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2) \right] \\ x_2 &= \frac{1}{2\pi} \arctan \frac{y_2}{y_1} \end{aligned} \tag{7.2.11}$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2} \right] \left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2} \right] \tag{7.2.12}$$

Since this is the product of a function of y_2 alone and a function of y_1 alone, we see that each y is independently distributed according to the normal distribution (7.2.9).

One further trick is useful in applying (7.2.10). Suppose that, instead of picking uniform deviates x_1 and x_2 in the unit square, we instead pick v_1 and v_2 as the ordinate and abscissa of a random point inside the unit circle around the origin. Then the sum of their squares, $R^2 \equiv v_1^2 + v_2^2$ is a uniform deviate, which can be used for x_1, while the angle that (v_1, v_2) defines with respect to the v_1 axis can serve as the random angle $2\pi x_2$. What's the advantage? It's that the cosine and sine in (7.2.10) can now be written as $v_1/\sqrt{R^2}$ and $v_2/\sqrt{R^2}$, obviating the trigonometric function calls!

We thus have

```
      FUNCTION gasdev(idum)
      INTEGER idum
      REAL gasdev
C     USES ran1
        Returns a normally distributed deviate with zero mean and unit variance, using ran1(idum)
        as the source of uniform deviates.
      INTEGER iset
      REAL fac,gset,rsq,v1,v2,ran1
      SAVE iset,gset
      DATA iset/0/
      if (idum.lt.0) iset=0                  Reinitialize.
      if (iset.eq.0) then                    We don't have an extra deviate handy, so
1       v1=2.*ran1(idum)-1.                  pick two uniform numbers in the square extend-
          v2=2.*ran1(idum)-1.                  ing from -1 to +1 in each direction,
          rsq=v1**2+v2**2                    see if they are in the unit circle,
          if(rsq.ge.1..or.rsq.eq.0.)goto 1   and if they are not, try again.
          fac=sqrt(-2.*log(rsq)/rsq)         Now make the Box-Muller transformation to get
          gset=v1*fac                            two normal deviates. Return one and save
          gasdev=v2*fac                          the other for next time.
          iset=1                             Set flag.
      else                                   We have an extra deviate handy,
          gasdev=gset                        so return it,
          iset=0                             and unset the flag.
      endif
      return
      END
```

See Devroye [1] and Bratley [2] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §9.1. [1]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [2]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 116ff.

7.3 Rejection Method: Gamma, Poisson, Binomial Deviates

The *rejection method* is a powerful, general technique for generating random deviates whose distribution function $p(x)dx$ (probability of a value occurring between x and $x + dx$) is known and computable. The rejection method does *not* require that the cumulative distribution function [indefinite integral of $p(x)$] be readily computable, much less the inverse of that function — which was required for the transformation method in the previous section.

The rejection method is based on a simple geometrical argument:

Draw a graph of the probability distribution $p(x)$ that you wish to generate, so that the area under the curve in any range of x corresponds to the desired probability of generating an x in that range. If we had some way of choosing a random point *in two dimensions*, with uniform probability in the *area* under your curve, then the x value of that random point would have the desired distribution.

Now, on the same graph, draw any other curve $f(x)$ which has finite (not infinite) area and lies everywhere *above* your original probability distribution. (This is always possible, because your original curve encloses only unit area, by definition of probability.) We will call this $f(x)$ the *comparison function*. Imagine now that you have some way of choosing a random point in two dimensions that is uniform in the area under the comparison function. Whenever that point lies outside the area under the original probability distribution, we will *reject* it and choose another random point. Whenever it lies inside the area under the original probability distribution, we will *accept* it. It should be obvious that the accepted points are uniform in the accepted area, so that their x values have the desired distribution. It should also be obvious that the fraction of points rejected just depends on the ratio of the area of the comparison function to the area of the probability distribution function, not on the details of shape of either function. For example, a comparison function whose area is less than 2 will reject fewer than half the points, even if it approximates the probability function very badly at some values of x, e.g., remains finite in some region where x is zero.

It remains only to suggest how to choose a uniform random point in two dimensions under the comparison function $f(x)$. A variant of the transformation method (§7.2) does nicely: Be sure to have chosen a comparison function whose indefinite integral is known analytically, and is also analytically invertible to give x as a function of "area under the comparison function to the left of x." Now pick a uniform deviate between 0 and A, where A is the total area under $f(x)$, and use it to get a corresponding x. Then pick a uniform deviate between 0 and $f(x)$ as the y value for the two-dimensional point. You should be able to convince yourself that the point (x, y) is uniformly distributed in the area under the comparison function $f(x)$.

An equivalent procedure is to pick the second uniform deviate between zero and one, and accept or reject according to whether it is respectively less than or greater than the ratio $p(x)/f(x)$.

So, to summarize, the rejection method for some given $p(x)$ requires that one find, once and for all, some reasonably good comparison function $f(x)$. Thereafter, each deviate generated requires two uniform random deviates, one evaluation of f (to get the coordinate y), and one evaluation of p (to decide whether to accept or reject

7.3 Rejection Method: Gamma, Poisson, Binomial Deviates

The *rejection method* is a powerful, general technique for generating random deviates whose distribution function $p(x)dx$ (probability of a value occurring between x and $x + dx$) is known and computable. The rejection method does *not* require that the cumulative distribution function [indefinite integral of $p(x)$] be readily computable, much less the inverse of that function — which was required for the transformation method in the previous section.

The rejection method is based on a simple geometrical argument:

Draw a graph of the probability distribution $p(x)$ that you wish to generate, so that the area under the curve in any range of x corresponds to the desired probability of generating an x in that range. If we had some way of choosing a random point *in two dimensions*, with uniform probability in the *area* under your curve, then the x value of that random point would have the desired distribution.

Now, on the same graph, draw any other curve $f(x)$ which has finite (not infinite) area and lies everywhere *above* your original probability distribution. (This is always possible, because your original curve encloses only unit area, by definition of probability.) We will call this $f(x)$ the *comparison function*. Imagine now that you have some way of choosing a random point in two dimensions that is uniform in the area under the comparison function. Whenever that point lies outside the area under the original probability distribution, we will *reject* it and choose another random point. Whenever it lies inside the area under the original probability distribution, we will *accept* it. It should be obvious that the accepted points are uniform in the accepted area, so that their x values have the desired distribution. It should also be obvious that the fraction of points rejected just depends on the ratio of the area of the comparison function to the area of the probability distribution function, not on the details of shape of either function. For example, a comparison function whose area is less than 2 will reject fewer than half the points, even if it approximates the probability function very badly at some values of x, e.g., remains finite in some region where x is zero.

It remains only to suggest how to choose a uniform random point in two dimensions under the comparison function $f(x)$. A variant of the transformation method (§7.2) does nicely: Be sure to have chosen a comparison function whose indefinite integral is known analytically, and is also analytically invertible to give x as a function of "area under the comparison function to the left of x." Now pick a uniform deviate between 0 and A, where A is the total area under $f(x)$, and use it to get a corresponding x. Then pick a uniform deviate between 0 and $f(x)$ as the y value for the two-dimensional point. You should be able to convince yourself that the point (x, y) is uniformly distributed in the area under the comparison function $f(x)$.

An equivalent procedure is to pick the second uniform deviate between zero and one, and accept or reject according to whether it is respectively less than or greater than the ratio $p(x)/f(x)$.

So, to summarize, the rejection method for some given $p(x)$ requires that one find, once and for all, some reasonably good comparison function $f(x)$. Thereafter, each deviate generated requires two uniform random deviates, one evaluation of f (to get the coordinate y), and one evaluation of p (to decide whether to accept or reject

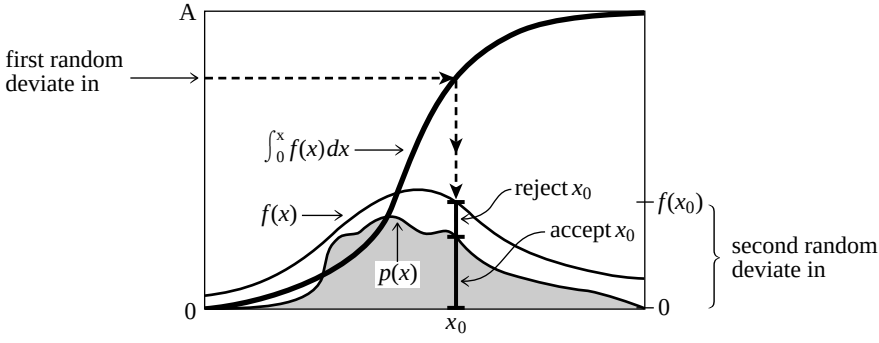


Figure 7.3.1. Rejection method for generating a random deviate x from a known probability distribution $p(x)$ that is everywhere less than some other function $f(x)$. The transformation method is first used to generate a random deviate x of the distribution f (compare Figure 7.2.1). A second uniform deviate is used to decide whether to accept or reject that x. If it is rejected, a new deviate of f is found; and so on. The ratio of accepted to rejected points is the ratio of the area under p to the area between p and f.

the point x, y). Figure 7.3.1 illustrates the procedure. Then, of course, this procedure must be repeated, on the average, A times before the final deviate is obtained.

Gamma Distribution

The gamma distribution of integer order $a > 0$ is the waiting time to the ath event in a Poisson random process of unit mean. For example, when $a = 1$, it is just the exponential distribution of §7.2, the waiting time to the first event.

A gamma deviate has probability $p_a(x)dx$ of occurring with a value between x and $x + dx$, where

$$p_a(x)dx = \frac{x^{a-1}e^{-x}}{\Gamma(a)}dx \qquad x > 0 \tag{7.3.1}$$

To generate deviates of (7.3.1) for small values of a, it is best to add up a exponentially distributed waiting times, i.e., logarithms of uniform deviates. Since the sum of logarithms is the logarithm of the product, one really has only to generate the product of a uniform deviates, then take the log.

For larger values of a, the distribution (7.3.1) has a typically "bell-shaped" form, with a peak at $x = a$ and a half-width of about $\sqrt{a}$.

We will be interested in several probability distributions with this same qualitative form. A useful comparison function in such cases is derived from the *Lorentzian distribution*

$$p(y)dy = \frac{1}{\pi}\left(\frac{1}{1+y^2}\right)dy \tag{7.3.2}$$

whose inverse indefinite integral is just the tangent function. It follows that the x-coordinate of an area-uniform random point under the comparison function

$$f(x) = \frac{c_0}{1 + (x - x_0)^2/a_0^2} \tag{7.3.3}$$

for any constants a_0, c_0, and x_0, can be generated by the prescription

$$x = a_0 \tan(\pi U) + x_0 \tag{7.3.4}$$

where U is a uniform deviate between 0 and 1. Thus, for some specific "bell-shaped" $p(x)$ probability distribution, we need only find constants a_0, c_0, x_0, with the product $a_0 c_0$ (which determines the area) as small as possible, such that (7.3.3) is everywhere greater than $p(x)$.

Ahrens has done this for the gamma distribution, yielding the following algorithm (as described in Knuth [1]):

```
      FUNCTION gamdev(ia,idum)
      INTEGER ia,idum
      REAL gamdev
C     USES ran1
        Returns a deviate distributed as a gamma distribution of integer order ia, i.e., a waiting
        time to the iath event in a Poisson process of unit mean, using ran1(idum) as the source
        of uniform deviates.
      INTEGER j
      REAL am,e,s,v1,v2,x,y,ran1
      if(ia.lt.1)pause 'bad argument in gamdev'
      if(ia.lt.6)then                 Use direct method, adding waiting times.
          x=1.
          do 11 j=1,ia
              x=x*ran1(idum)
          enddo 11
          x=-log(x)
      else                            Use rejection method.
1             v1=ran1(idum)           These four lines generate the tangent of a random angle, i.e.,
              v2=2.*ran1(idum)-1.       are equivalent to y = tan(3.14159265 * ran1(idum)).
          if(v1**2+v2**2.gt.1.)goto 1
              y=v2/v1
              am=ia-1
              s=sqrt(2.*am+1.)
              x=s*y+am                We decide whether to reject x:
          if(x.le.0.)goto 1           Reject in region of zero probability.
              e=(1.+y**2)*exp(am*log(x/am)-s*y)     Ratio of prob. fn. to comparison fn.
          if(ran1(idum).gt.e)goto 1                 Reject on basis of a second uniform de-
      endif                                           viate.
      gamdev=x
      return
      END
```

Poisson Deviates

The Poisson distribution is conceptually related to the gamma distribution. It gives the probability of a certain integer number m of unit rate Poisson random events occurring in a given interval of time x, while the gamma distribution was the probability of waiting time between x and $x + dx$ to the mth event. Note that m takes on only integer values ≥ 0, so that the Poisson distribution, viewed as a continuous distribution function $p_x(m)dm$, is zero everywhere except where m is an integer ≥ 0. At such places, it is infinite, such that the integrated probability over a region containing the integer is some finite number. The total probability at an integer j is

$$\text{Prob}(j) = \int_{j-\epsilon}^{j+\epsilon} p_x(m)dm = \frac{x^j e^{-x}}{j!} \tag{7.3.5}$$

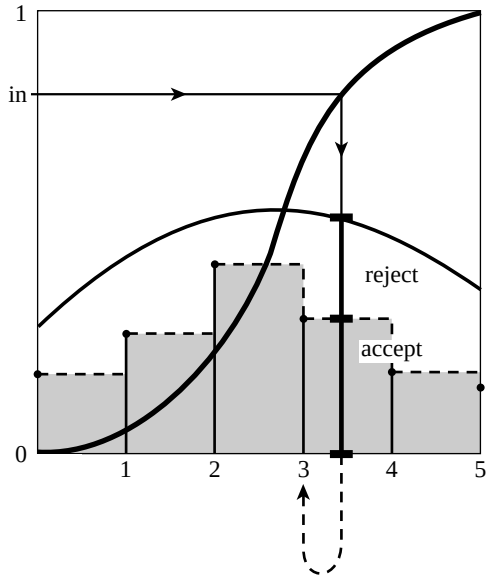


Figure 7.3.2. Rejection method as applied to an integer-valued distribution. The method is performed on the step function shown as a dashed line, yielding a real-valued deviate. This deviate is rounded down to the next lower integer, which is output.

At first sight this might seem an unlikely candidate distribution for the rejection method, since no continuous comparison function can be larger than the infinitely tall, but infinitely narrow, *Dirac delta functions* in $p_x(m)$. However, there is a trick that we can do: Spread the finite area in the spike at j uniformly into the interval between j and $j+1$. This defines a continuous distribution $q_x(m)dm$ given by

$$q_x(m)dm = \frac{x^{[m]}e^{-x}}{[m]!}dm \tag{7.3.6}$$

where $[m]$ represents the largest integer less than m. If we now use the rejection method to generate a (noninteger) deviate from (7.3.6), and then take the integer part of that deviate, it will be as if drawn from the desired distribution (7.3.5). (See Figure 7.3.2.) This trick is general for any integer-valued probability distribution.

For x large enough, the distribution (7.3.6) is qualitatively bell-shaped (albeit with a bell made out of small, square steps), and we can use the same kind of Lorentzian comparison function as was already used above. For small x, we can generate independent exponential deviates (waiting times between events); when the sum of these first exceeds x, then the number of events that would have occurred in waiting time x becomes known and is one less than the number of terms in the sum.

These ideas produce the following routine:

```
      FUNCTION poidev(xm,idum)
      INTEGER idum
      REAL poidev,xm,PI
      PARAMETER (PI=3.141592654)
C     USES gammln,ran1
```

Returns as a floating-point number an integer value that is a random deviate drawn from a Poisson distribution of mean `xm`, using `ran1(idum)` as a source of uniform random deviates.

```
      REAL alxm,em,g,oldm,sq,t,y,gammln,ran1
      SAVE alxm,g,oldm,sq
      DATA oldm /-1./               Flag for whether xm has changed since last call.
      if (xm.lt.12.)then            Use direct method.
          if (xm.ne.oldm) then
              oldm=xm
              g=exp(-xm)            If xm is new, compute the exponential.
          endif
          em=-1
          t=1.
2         em=em+1.                  Instead of adding exponential deviates it is equivalent to mul-
          t=t*ran1(idum)                tiply uniform deviates. We never actually have to take the
          if (t.gt.g) goto 2            log, merely compare to the pre-computed exponential.
      else                          Use rejection method.
          if (xm.ne.oldm) then      If xm has changed since the last call, then precompute some
              oldm=xm                   functions that occur below.
              sq=sqrt(2.*xm)
              alxm=log(xm)
              g=xm*alxm-gammln(xm+1.) The function gammln is the natural log of the gamma
          endif                             function, as given in §6.1.
1         y=tan(PI*ran1(idum))      y is a deviate from a Lorentzian comparison function.
          em=sq*y+xm                em is y, shifted and scaled.
          if (em.lt.0.) goto 1      Reject if in regime of zero probability.
          em=int(em)                The trick for integer-valued distributions.
          t=0.9*(1.+y**2)*exp(em*alxm-gammln(em+1.)-g)     The ratio of the desired distribu-
          if (ran1(idum).gt.t) goto 1       tion to the comparison function; we accept or re-
      endif                                 ject by comparing it to another uniform deviate.
      poidev=em                             The factor 0.9 is chosen so that t never exceeds
      return                                1.
      END
```

Binomial Deviates

If an event occurs with probability q, and we make n trials, then the number of times m that it occurs has the binomial distribution,

$$\int_{j-\epsilon}^{j+\epsilon} p_{n,q}(m)dm = \binom{n}{j} q^j (1-q)^{n-j} \tag{7.3.7}$$

The binomial distribution is integer valued, with m taking on possible values from 0 to n. It depends on *two* parameters, n and q, so is correspondingly a bit harder to implement than our previous examples. Nevertheless, the techniques already illustrated are sufficiently powerful to do the job:

```
      FUNCTION bnldev(pp,n,idum)
      INTEGER idum,n
      REAL bnldev,pp,PI
C     USES gammln,ran1
      PARAMETER (PI=3.141592654)
          Returns as a floating-point number an integer value that is a random deviate drawn from
          a binomial distribution of n trials each of probability pp, using ran1(idum) as a source
          of uniform random deviates.
      INTEGER j,nold
      REAL am,em,en,g,oldg,p,pc,pclog,plog,pold,sq,t,y,gammln,ran1
```

```
      SAVE nold,pold,pc,plog,pclog,en,oldg
      DATA nold /-1/, pold /-1./  Arguments from previous calls.
      if(pp.le.0.5)then           The binomial distribution is invariant under changing pp to
          p=pp                        1.-pp, if we also change the answer to n minus itself;
      else                            we'll remember to do this below.
          p=1.-pp
      endif
      am=n*p                      This is the mean of the deviate to be produced.
      if (n.lt.25)then            Use the direct method while n is not too large. This can
          bnldev=0.                   require up to 25 calls to ran1.
          do 11 j=1,n
              if(ran1(idum).lt.p)bnldev=bnldev+1.
          enddo 11
      else if (am.lt.1.) then     If fewer than one event is expected out of 25 or more tri-
          g=exp(-am)                  als, then the distribution is quite accurately Poisson. Use
          t=1.                        direct Poisson method.
          do 12 j=0,n
              t=t*ran1(idum)
              if (t.lt.g) goto 1
          enddo 12
          j=n
1         bnldev=j
      else                        Use the rejection method.
          if (n.ne.nold) then     If n has changed, then compute useful quantities.
              en=n
              oldg=gammln(en+1.)
              nold=n
          endif
          if (p.ne.pold) then     If p has changed, then compute useful quantities.
              pc=1.-p
              plog=log(p)
              pclog=log(pc)
              pold=p
          endif
          sq=sqrt(2.*am*pc)       The following code should by now seem familiar: rejection
2         y=tan(PI*ran1(idum))        method with a Lorentzian comparison function.
          em=sq*y+am
          if (em.lt.0..or.em.ge.en+1.) goto 2       Reject.
          em=int(em)                                Trick for integer-valued distribution.
          t=1.2*sq*(1.+y**2)*exp(oldg-gammln(em+1.)
     *      -gammln(en-em+1.)+em*plog+(en-em)*pclog)
          if (ran1(idum).gt.t) goto 2     Reject. This happens about 1.5 times per deviate, on
          bnldev=em                           average.
      endif
      if (p.ne.pp) bnldev=n-bnldev        Remember to undo the symmetry transformation.
      return
      END
```

See Devroye [2] and Bratley [3] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 120ff. [1]

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §X.4. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3].

7.4 Generation of Random Bits

This topic is not very useful for programming in high-level languages, but it can be quite useful when you have access to the machine-language level of a machine or when you are in a position to build special-purpose hardware out of readily available chips.

The problem is how to generate single random bits, with 0 and 1 equally probable. Of course you can just generate uniform random deviates between zero and one and use their high-order bit (i.e., test if they are greater than or less than 0.5). However this takes a lot of arithmetic; there are special-purpose applications, such as real-time signal processing, where you want to generate bits very much faster than that.

One method for generating random bits, with two variant implementations, is based on "primitive polynomials modulo 2." The theory of these polynomials is beyond our scope (although §7.7 and §20.3 will give you small tastes of it). Here, suffice it to say that there are special polynomials among those whose coefficients are zero or one. An example is

$$x^{18} + x^5 + x^2 + x^1 + x^0 \tag{7.4.1}$$

which we can abbreviate by just writing the nonzero powers of x, e.g.,

$$(18, 5, 2, 1, 0)$$

Every primitive polynomial modulo 2 of order n (=18 above) defines a recurrence relation for obtaining a new random bit from the n preceding ones. The recurrence relation is guaranteed to produce a sequence of maximal length, i.e., cycle through all possible sequences of n bits (except all zeros) before it repeats. Therefore one can seed the sequence with any initial bit pattern (except all zeros), and get $2^n - 1$ random bits before the sequence repeats.

Let the bits be numbered from 1 (most recently generated) through n (generated n steps ago), and denoted $a_1, a_2, \ldots, a_n$. We want to give a formula for a new bit a_0. After generating a_0 we will shift all the bits by one, so that the old a_n is finally lost, and the new a_0 becomes a_1. We then apply the formula again, and so on.

"Method I" is the easiest to implement in hardware, requiring only a single shift register n bits long and a few XOR ("exclusive or" or bit addition mod 2) gates. For the primitive polynomial given above, the recurrence formula is

$$a_0 = a_{18} \text{ XOR } a_5 \text{ XOR } a_2 \text{ XOR } a_1 \tag{7.4.2}$$

The terms that are XOR'd together can be thought of as "taps" on the shift register, XOR'd into the register's input. More generally, there is precisely one term for each nonzero coefficient in the primitive polynomial except the constant (zero bit) term. So the first term will always be a_n for a primitive polynomial of degree n, while the last term might or might not be a_1, depending on whether the primitive polynomial has a term in x^1.

It is rather cumbersome to illustrate the method in `FORTRAN`. Assume that `iand` is a bitwise AND function, `not` is bitwise complement, `ishft( ,1)` is leftshift by one bit, `ior` is bitwise OR. (These are available in many `FORTRAN` implementations.) Then we have the following routine.

7.4 Generation of Random Bits

This topic is not very useful for programming in high-level languages, but it can be quite useful when you have access to the machine-language level of a machine or when you are in a position to build special-purpose hardware out of readily available chips.

The problem is how to generate single random bits, with 0 and 1 equally probable. Of course you can just generate uniform random deviates between zero and one and use their high-order bit (i.e., test if they are greater than or less than 0.5). However this takes a lot of arithmetic; there are special-purpose applications, such as real-time signal processing, where you want to generate bits very much faster than that.

One method for generating random bits, with two variant implementations, is based on "primitive polynomials modulo 2." The theory of these polynomials is beyond our scope (although §7.7 and §20.3 will give you small tastes of it). Here, suffice it to say that there are special polynomials among those whose coefficients are zero or one. An example is

$$x^{18} + x^5 + x^2 + x^1 + x^0 \tag{7.4.1}$$

which we can abbreviate by just writing the nonzero powers of x, e.g.,

$$(18, 5, 2, 1, 0)$$

Every primitive polynomial modulo 2 of order n (=18 above) defines a recurrence relation for obtaining a new random bit from the n preceding ones. The recurrence relation is guaranteed to produce a sequence of maximal length, i.e., cycle through all possible sequences of n bits (except all zeros) before it repeats. Therefore one can seed the sequence with any initial bit pattern (except all zeros), and get $2^n - 1$ random bits before the sequence repeats.

Let the bits be numbered from 1 (most recently generated) through n (generated n steps ago), and denoted $a_1, a_2, \ldots, a_n$. We want to give a formula for a new bit a_0. After generating a_0 we will shift all the bits by one, so that the old a_n is finally lost, and the new a_0 becomes a_1. We then apply the formula again, and so on.

"Method I" is the easiest to implement in hardware, requiring only a single shift register n bits long and a few XOR ("exclusive or" or bit addition mod 2) gates. For the primitive polynomial given above, the recurrence formula is

$$a_0 = a_{18} \text{ XOR } a_5 \text{ XOR } a_2 \text{ XOR } a_1 \tag{7.4.2}$$

The terms that are XOR'd together can be thought of as "taps" on the shift register, XOR'd into the register's input. More generally, there is precisely one term for each nonzero coefficient in the primitive polynomial except the constant (zero bit) term. So the first term will always be a_n for a primitive polynomial of degree n, while the last term might or might not be a_1, depending on whether the primitive polynomial has a term in x^1.

It is rather cumbersome to illustrate the method in `FORTRAN`. Assume that `iand` is a bitwise AND function, `not` is bitwise complement, `ishft( ,1)` is leftshift by one bit, `ior` is bitwise OR. (These are available in many `FORTRAN` implementations.) Then we have the following routine.

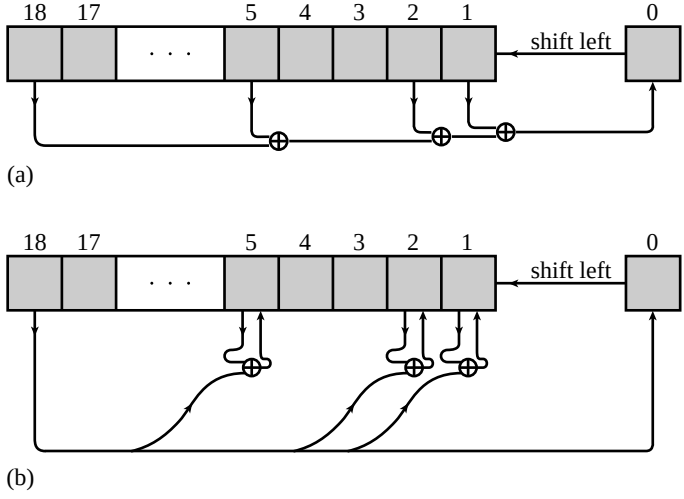


Figure 7.4.1. Two related methods for obtaining random bits from a shift register and a primitive polynomial modulo 2. (a) The contents of selected taps are combined by exclusive-or (addition modulo 2), and the result is shifted in from the right. This method is easiest to implement in hardware. (b) Selected bits are modified by exclusive-or with the leftmost bit, which is then shifted in from the right. This method is easiest to implement in software.

```
FUNCTION irbit1(iseed)
INTEGER irbit1,iseed,IB1,IB2,IB5,IB18
PARAMETER (IB1=1,IB2=2,IB5=16,IB18=131072)        Powers of 2.
    Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which
    is modified for the next call).
LOGICAL newbit                                    The accumulated XOR's.
newbit=iand(iseed,IB18).ne.0                      Get bit 18.
if(iand(iseed,IB5).ne.0)newbit=.not.newbit        XOR with bit 5.
if(iand(iseed,IB2).ne.0)newbit=.not.newbit        XOR with bit 2.
if(iand(iseed,IB1).ne.0)newbit=.not.newbit        XOR with bit 1.
irbit1=0
iseed=iand(ishft(iseed,1),not(IB1))    Leftshift the seed and put a zero in its bit 1.
if(newbit)then                         But if the XOR calculation gave a 1,
    irbit1=1
    iseed=ior(iseed,IB1)               then put that in bit 1 instead.
endif
return
END
```

"Method II" is less suited to direct hardware implementation (though still possible), but is more suited to machine-language implementation. It modifies more than one bit among the saved n bits as each new bit is generated (Figure 7.4.1). It generates the maximal length sequence, but not in the same order as Method I. The prescription for the primitive polynomial (7.4.1) is:

$$\begin{aligned} a_0 &= a_{18} \\ a_5 &= a_5 \text{ XOR } a_0 \\ a_2 &= a_2 \text{ XOR } a_0 \\ a_1 &= a_1 \text{ XOR } a_0 \end{aligned} \tag{7.4.3}$$

Some Primitive Polynomials Modulo 2 (after Watson)

(1, 0)	(51, 6, 3, 1, 0)
(2, 1, 0)	(52, 3, 0)
(3, 1, 0)	(53, 6, 2, 1, 0)
(4, 1, 0)	(54, 6, 5, 4, 3, 2, 0)
(5, 2, 0)	(55, 6, 2, 1, 0)
(6, 1, 0)	(56, 7, 4, 2, 0)
(7, 1, 0)	(57, 5, 3, 2, 0)
(8, 4, 3, 2, 0)	(58, 6, 5, 1, 0)
(9, 4, 0)	(59, 6, 5, 4, 3, 1, 0)
(10, 3, 0)	(60, 1, 0)
(11, 2, 0)	(61, 5, 2, 1, 0)
(12, 6, 4, 1, 0)	(62, 6, 5, 3, 0)
(13, 4, 3, 1, 0)	(63, 1, 0)
(14, 5, 3, 1, 0)	(64, 4, 3, 1, 0)
(15, 1, 0)	(65, 4, 3, 1, 0)
(16, 5, 3, 2, 0)	(66, 8, 6, 5, 3, 2, 0)
(17, 3, 0)	(67, 5, 2, 1, 0)
(18, 5, 2, 1, 0)	(68, 7, 5, 1, 0)
(19, 5, 2, 1, 0)	(69, 6, 5, 2, 0)
(20, 3, 0)	(70, 5, 3, 1, 0)
(21, 2, 0)	(71, 5, 3, 1, 0)
(22, 1, 0)	(72, 6, 4, 3, 2, 1, 0)
(23, 5, 0)	(73, 4, 3, 2, 0)
(24, 4, 3, 1, 0)	(74, 7, 4, 3, 0)
(25, 3, 0)	(75, 6, 3, 1, 0)
(26, 6, 2, 1, 0)	(76, 5, 4, 2, 0)
(27, 5, 2, 1, 0)	(77, 6, 5, 2, 0)
(28, 3, 0)	(78, 7, 2, 1, 0)
(29, 2, 0)	(79, 4, 3, 2, 0)
(30, 6, 4, 1, 0)	(80, 7, 5, 3, 2, 1, 0)
(31, 3, 0)	(81, 4 0)
(32, 7, 5, 3, 2, 1, 0)	(82, 8, 7, 6, 4, 1, 0)
(33, 6, 4, 1, 0)	(83, 7, 4, 2, 0)
(34, 7, 6, 5, 2, 1, 0)	(84, 8, 7, 5, 3, 1, 0)
(35, 2, 0)	(85, 8, 2, 1, 0)
(36, 6, 5, 4, 2, 1, 0)	(86, 6, 5, 2, 0)
(37, 5, 4, 3, 2, 1, 0)	(87, 7, 5, 1, 0)
(38, 6, 5, 1, 0)	(88, 8, 5, 4, 3, 1, 0)
(39, 4, 0)	(89, 6, 5, 3, 0)
(40, 5, 4 3, 0)	(90, 5, 3, 2, 0)
(41, 3, 0)	(91, 7, 6, 5, 3, 2, 0)
(42, 5, 4, 3, 2, 1, 0)	(92, 6, 5, 2, 0)
(43, 6, 4, 3, 0)	(93, 2, 0)
(44, 6, 5, 2, 0)	(94, 6, 5, 1, 0)
(45, 4, 3, 1, 0)	(95, 6, 5, 4, 2, 1, 0)
(46, 8, 5, 3, 2, 1, 0)	(96, 7, 6, 4, 3, 2, 0)
(47, 5, 0)	(97, 6, 0)
(48, 7, 5, 4, 2, 1, 0)	(98, 7, 4, 3, 2, 1, 0)
(49, 6, 5, 4, 0)	(99, 7, 5, 4, 0)
(50, 4, 3, 2, 0)	(100, 8, 7, 2, 0)

In general there will be an exclusive-or for each nonzero term in the primitive polynomial except 0 and n. The nice feature about Method II is that all the exclusive-or's can usually be done as a single masked word XOR (here assumed to be the `FORTRAN` function `ieor`):

```
FUNCTION irbit2(iseed)
INTEGER irbit2,iseed,IB1,IB2,IB5,IB18,MASK
PARAMETER (IB1=1,IB2=2,IB5=16,IB18=131072,MASK=IB1+IB2+IB5)
    Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which
    is modified for the next call).
if(iand(iseed,IB18).ne.0)then   Change all masked bits, shift, and put 1 into bit 1.
    iseed=ior(ishft(ieor(iseed,MASK),1),IB1)
    irbit2=1
else                            Shift and put 0 into bit 1.
    iseed=iand(ishft(iseed,1),not(IB1))
    irbit2=0
endif
return
END
```

A word of caution is: Don't use sequential bits from these routines as the bits of a large, supposedly random, integer, or as the bits in the mantissa of a supposedly random floating-point number. They are not very random for that purpose; see Knuth [1]. Examples of acceptable uses of these random bits are: (i) multiplying a signal randomly by ± 1 at a rapid "chip rate," so as to spread its spectrum uniformly (but recoverably) across some desired bandpass, or (ii) Monte Carlo exploration of a binary tree, where decisions as to whether to branch left or right are to be made randomly.

Now we do not want you to go through life thinking that there is something special about the primitive polynomial of degree 18 used in the above examples. (We chose 18 because 2^{18} is small enough for you to verify our claims directly by numerical experiment.) The accompanying table [2] lists one primitive polynomial for each degree up to 100. (In fact there exist many such for each degree. For example, see §7.7 for a complete table up to degree 10.)

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 29ff. [1]

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (Cambridge: Cambridge University Press), §§9.32–9.37.

Tausworthe, R.C. 1965, *Mathematics of Computation*, vol. 19, pp. 201–209.

Watson, E.J. 1962, *Mathematics of Computation*, vol. 16, pp. 368–369. [2]

7.5 Random Sequences Based on Data Encryption

In *Numerical Recipes'* first edition, we described how to use the Data Encryption Standard (DES) [1-3] for the generation of random numbers. Unfortunately, when implemented in software in a high-level language like FORTRAN, DES is very slow, so excruciatingly slow, in fact, that our previous implementation can be viewed as more mischievous than useful. Here we give a much faster and simpler algorithm which, though it may not be secure in the cryptographic sense, generates about equally good random numbers.

DES, like its progenitor cryptographic system LUCIFER, is a so-called "block product cipher" [4]. It acts on 64 bits of input by iteratively applying (16 times, in fact) a kind of highly

```
FUNCTION irbit2(iseed)
INTEGER irbit2,iseed,IB1,IB2,IB5,IB18,MASK
PARAMETER (IB1=1,IB2=2,IB5=16,IB18=131072,MASK=IB1+IB2+IB5)
    Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which
    is modified for the next call).
if(iand(iseed,IB18).ne.0)then   Change all masked bits, shift, and put 1 into bit 1.
    iseed=ior(ishft(ieor(iseed,MASK),1),IB1)
    irbit2=1
else                            Shift and put 0 into bit 1.
    iseed=iand(ishft(iseed,1),not(IB1))
    irbit2=0
endif
return
END
```

A word of caution is: Don't use sequential bits from these routines as the bits of a large, supposedly random, integer, or as the bits in the mantissa of a supposedly random floating-point number. They are not very random for that purpose; see Knuth [1]. Examples of acceptable uses of these random bits are: (i) multiplying a signal randomly by ± 1 at a rapid "chip rate," so as to spread its spectrum uniformly (but recoverably) across some desired bandpass, or (ii) Monte Carlo exploration of a binary tree, where decisions as to whether to branch left or right are to be made randomly.

Now we do not want you to go through life thinking that there is something special about the primitive polynomial of degree 18 used in the above examples. (We chose 18 because 2^{18} is small enough for you to verify our claims directly by numerical experiment.) The accompanying table [2] lists one primitive polynomial for each degree up to 100. (In fact there exist many such for each degree. For example, see §7.7 for a complete table up to degree 10.)

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 29ff. [1]

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (Cambridge: Cambridge University Press), §§9.32–9.37.

Tausworthe, R.C. 1965, *Mathematics of Computation*, vol. 19, pp. 201–209.

Watson, E.J. 1962, *Mathematics of Computation*, vol. 16, pp. 368–369. [2]

7.5 Random Sequences Based on Data Encryption

In *Numerical Recipes'* first edition, we described how to use the Data Encryption Standard (DES) [1-3] for the generation of random numbers. Unfortunately, when implemented in software in a high-level language like FORTRAN, DES is very slow, so excruciatingly slow, in fact, that our previous implementation can be viewed as more mischievous than useful. Here we give a much faster and simpler algorithm which, though it may not be secure in the cryptographic sense, generates about equally good random numbers.

DES, like its progenitor cryptographic system LUCIFER, is a so-called "block product cipher" [4]. It acts on 64 bits of input by iteratively applying (16 times, in fact) a kind of highly

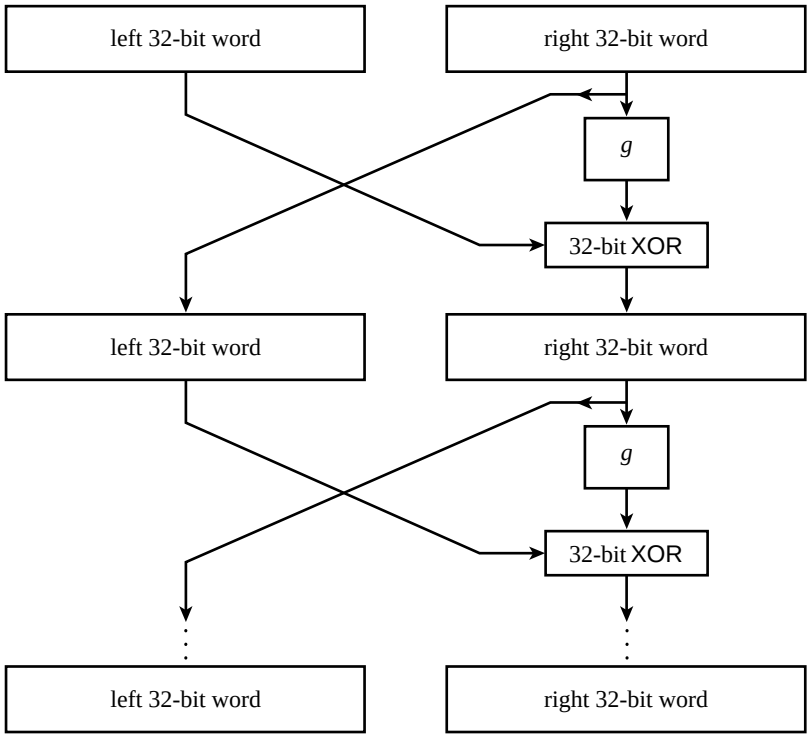


Figure 7.5.1. The Data Encryption Standard (DES) iterates a nonlinear function g on two 32-bit words, in the manner shown here (after Meyer and Matyas [4]).

nonlinear bit-mixing function. Figure 7.5.1 shows the flow of information in DES during this mixing. The function g, which takes 32-bits into 32-bits, is called the "cipher function." Meyer and Matyas [4] discuss the importance of the cipher function being nonlinear, as well as other design criteria.

DES constructs its cipher function g from an intricate set of bit permutations and table lookups acting on short sequences of consecutive bits. Apparently, this function was chosen to be particularly strong cryptographically (or conceivably as some critics contend, to have an exquisitely subtle cryptographic flaw!). For our purposes, a different function g that can be rapidly computed in a high-level computer language is preferable. Such a function may weaken the algorithm cryptographically. Our purposes are not, however, cryptographic: We want to find the fastest g, and smallest number of iterations of the mixing procedure in Figure 7.5.1, such that our output random sequence passes the standard tests that are customarily applied to random number generators. The resulting algorithm will not be DES, but rather a kind of "pseudo-DES," better suited to the purpose at hand.

Following the criterion, mentioned above, that g should be nonlinear, we must give the integer multiply operation a prominent place in g. Because 64-bit registers are not generally accessible in high-level languages, we must confine ourselves to multiplying 16-bit operands into a 32-bit result. So, the general idea of g, almost forced, is to calculate the three distinct 32-bit products of the high and low 16-bit input half-words, and then to combine these, and perhaps additional fixed constants, by fast operations (e.g., add or exclusive-or) into a single 32-bit result.

There are only a limited number of ways of effecting this general scheme, allowing systematic exploration of the alternatives. Experimentation, and tests of the randomness of the output, lead to the sequence of operations shown in Figure 7.5.2. The few new elements in the figure need explanation: The values C_1 and C_2 are fixed constants, chosen randomly with the constraint that they have exactly 16 1-bits and 16 0-bits; combining these constants

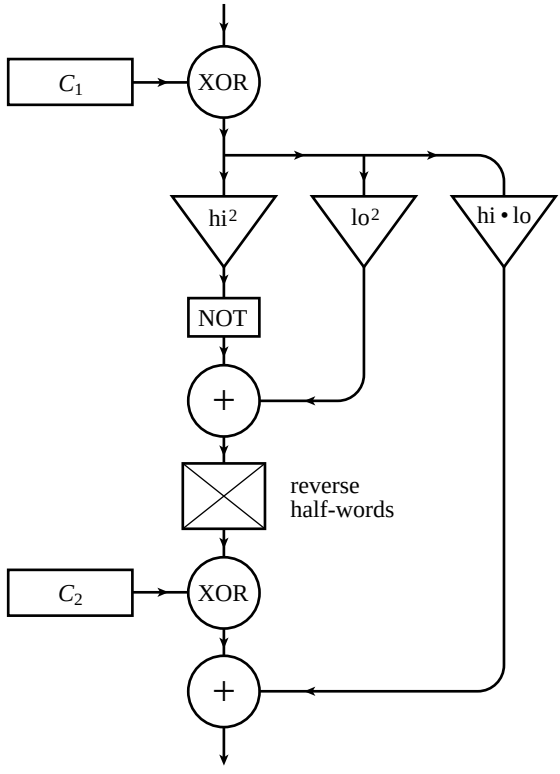


Figure 7.5.2. The nonlinear function g used by the routine `psdes`.

via exclusive-or ensures that the overall g has no bias towards 0 or 1 bits.

The "reverse half-words" operation in Figure 7.5.2 turns out to be essential; otherwise, the very lowest and very highest bits are not properly mixed by the three multiplications. The nonobvious choices in g are therefore: where along the vertical "pipeline" to do the reverse; in what order to combine the three products and C_2; and with which operation (add or exclusive-or) should each combining be done? We tested these choices exhaustively before settling on the algorithm shown in the figure.

It remains to determine the smallest number of iterations N_{it} that we can get away with. The minimum meaningful N_{it} is evidently two, since a single iteration simply moves one 32-bit word without altering it. One can use the constants C_1 and C_2 to help determine an appropriate N_{it}: When $N_{it} = 2$ and $C_1 = C_2 = 0$ (an intentionally very poor choice), the generator fails several tests of randomness by easily measurable, though not overwhelming, amounts. When $N_{it} = 4$, on the other hand, or with $N_{it} = 2$ but with the constants C_1, C_2 nonsparse, we have been unable to find *any* statistical deviation from randomness in sequences of up to 10^9 floating numbers r_i derived from this scheme. The combined strength of $N_{it} = 4$ and nonsparse C_1, C_2 should therefore give sequences that are random to tests even far beyond those that we have actually tried. These are our recommended conservative parameter values, notwithstanding the fact that $N_{it} = 2$ (which is, of course, twice as fast) has no nonrandomness discernible (by us).

We turn now to implementation. The nonlinear function shown in Figure 7.5.2 is not implementable in strictly portable `FORTRAN`, for at least three reasons: (1) The addition of two 32-bit integers may overflow, and the multiplication of two 16-bit integers may not produce the correct 32-bit product because of sign-bit conventions. We intend that the overflow be ignored, and that the 16-bit integers be multiplied as if they are positive. It is possible to force this behavior on most machines. (2) We assume 32-bit integers; however, there

is no reason to believe that *longer* integers would be in any way inferior (with suitable extensions of the constants C_1, C_2). (3) Your compiler may require a different notation for hex constants (see below).

We have been able to run the following routine, `psdes`, successfully on machines ranging from PCs to VAXes and both "big-endian" and "little-endian" UNIX workstations. (Big- and little-endian refer to the order in which the bytes are stored in a word.) A strictly portable implementation is possible in `C`. If all else fails, you can make a `FORTRAN`-callable version of the `C` routine, found in *Numerical Recipes in C*.

```
      SUBROUTINE psdes(lword,irword)
      INTEGER irword,lword,NITER
      PARAMETER (NITER=4)
          "Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are
          returned hashed on all bits. NOTE: This routine assumes that arbitrary 32-bit integers can
          be added without overflow. To accomplish this, you may need to compile with a special
          directive (e.g., /check=nooverflow for VMS). In other languages, such as C, one can
          instead type the integers as "unsigned."
      INTEGER i,ia,ib,iswap,itmph,itmpl,c1(4),c2(4)
      SAVE c1,c2
      DATA c1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',        Your compiler may use a differ-
     *     Z'0F33D1B2'/, c2 /Z'4B0F3B58',Z'E874F0C3',          ent notation for hex constants!
     *     Z'6955C5A6', Z'55A7CA46'/
      do 11 i=1,NITER                  Perform niter iterations of DES logic, using a simpler (non-
          iswap=irword                     cryptographic) nonlinear function instead of DES's.
          ia=ieor(irword,c1(i))        The bit-rich constants c1 and (below) c2 guarantee lots of
          itmpl=iand(ia,65535)             nonlinear mixing.
          itmph=iand(ishft(ia,-16),65535)
          ib=itmpl**2+not(itmph**2)
          ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
          irword=ieor(lword,ieor(c2(i),ia)+itmpl*itmph)
          lword=iswap
      enddo 11
      return
      END
```

The routine `ran4`, listed below, uses `psdes` to generate uniform random deviates. We adopt the convention that a negative value of the argument `idum` sets the left 32-bit word, while a positive value i sets the right 32-bit word, returns the ith random deviate, and increments `idum` to $i+1$. This is no more than a convenient way of defining many different sequences (negative values of `idum`), but still with random access to each sequence (positive values of `idum`). For getting a floating-point number from the 32-bit integer, we like to do it by the masking trick described at the end of §7.1, above. The hex constants 3F800000 and 007FFFFF are the appropriate ones for computers using the IEEE representation for 32-bit floating-point numbers (e.g., IBM PCs and most UNIX workstations). For DEC VAXes, the correct hex constants are, respectively, 00004080 and FFFF007F. Note that your compiler may require a different notation for hex constants, e.g., `x'3f800000'`, `'3F800000'X`, or even `16#3F800000`. For greater portability, you can instead construct a floating number by making the (signed) 32-bit integer nonnegative (typically, you add exactly 2^{31} if it is negative) and then multiplying it by a floating constant (typically $2.^{-31}$).

An interesting, and sometimes useful, feature of the routine `ran4`, below, is that it allows random access to the nth random value in a sequence, without the necessity of first generating values $1 \cdots n-1$. This property is shared by any random number generator based on *hashing* (the technique of mapping data keys, which may be highly clustered in value, approximately uniformly into a storage address space) [5,6]. One might have a simulation problem in which some certain rare situation becomes recognizable by its consequences only considerably after it has occurred. One may wish to restart the simulation back at that occurrence, using identical random values but, say, varying some other control parameters. The relevant question might then be something like "what random numbers were used in cycle number 337098901?" It might already be cycle number 395100273 before the question comes up. Random generators based on recursion, rather than hashing, cannot easily answer such a question.

Values for Verifying the Implementation of psdes						
idum	before psdes call		after psdes call (hex)		ran4(idum)	
	lword	irword	lword	irword	VAX	PC
–1	1	1	604D1DCE	509C0C23	0.275898	0.219120
99	1	99	D97F8571	A66CB41A	0.208204	0.849246
–99	99	1	7822309D	64300984	0.034307	0.375290
99	99	99	D7F376F0	59BA89EB	0.838676	0.457334

Successive calls to psdes with arguments -1, 99, -99, and 1, should produce exactly the lword and irword values shown. Masking conversion to a returned floating random value is allowed to be machine dependent; values for VAX and PC are shown.

```
      FUNCTION ran4(idum)
      INTEGER idum
      REAL ran4
C     USES psdes
        Returns a uniform random deviate in the range 0.0 to 1.0, generated by pseudo-DES (DES-
        like) hashing of the 64-bit word (idums,idum), where idums was set by a previous call with
        negative idum. Also increments idum. Routine can be used to generate a random sequence
        by successive calls, leaving idum unaltered between calls; or it can randomly access the nth
        deviate in a sequence by calling with idum = n. Different sequences are initialized by calls
        with differing negative values of idum.
      INTEGER idums,irword,itemp,jflmsk,jflone,lword
      REAL ftemp
      EQUIVALENCE (itemp,ftemp)
      SAVE idums,jflone,jflmsk
      DATA idums /0/, jflone /Z'3F800000'/, jflmsk /Z'007FFFFF'/
        The hexadecimal constants jflone and jflmsk are used to produce a floating number between
        1. and 2. by bitwise masking. They are machine-dependent. See text.
      if(idum.lt.0)then                       Reset idums and prepare to return the first devi-
          idums=-idum                           ate in its sequence.
          idum=1
      endif
      irword=idum
      lword=idums
      call psdes(lword,irword)                "Pseudo-DES" encode the words.
      itemp=ior(jflone,iand(jflmsk,irword))   Mask to a floating number between 1 and 2.
      ran4=ftemp-1.0                          Subtraction moves range to 0. to 1.
      idum=idum+1
      return
      END
```

The accompanying table gives data for verifying that ran4 and psdes work correctly on your machine. We do not advise the use of ran4 unless you are able to reproduce the hex values shown. Typically, ran4 is about 4 times slower than ran0 (§7.1), or about 3 times slower than ran1.

CITED REFERENCES AND FURTHER READING:

Data Encryption Standard, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards). [1]

Guidelines for Implementing and Using the NBS Data Encryption Standard, 1981 April 1, Federal Information Processing Standards Publication, number 74 (Washington: U.S. Department of Commerce, National Bureau of Standards). [2]

Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard, 1980, NBS Special Publication 500–20 (Washington: U.S. Department of Commerce, National Bureau of Standards). [3]

Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley). [4]

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 6. [5]

Vitter, J.S., and Chen, W-C. 1987, *Design and Analysis of Coalesced Hashing* (New York: Oxford University Press). [6]

7.6 Simple Monte Carlo Integration

Inspirations for numerical methods can spring from unlikely sources. "Splines" first were flexible strips of wood used by draftsmen. "Simulated annealing" (we shall see in §10.9) is rooted in a thermodynamic analogy. And who does not feel at least a faint echo of glamor in the name "Monte Carlo method"?

Suppose that we pick N random points, uniformly distributed in a multidimensional volume V. Call them $x_1, \ldots, x_N$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f \, dV \approx V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{7.6.1}$$

Here the angle brackets denote taking the arithmetic mean over the N sample points,

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(x_i) \tag{7.6.2}$$

The "plus-or-minus" term in (7.6.1) is a one standard deviation error estimate for the integral, not a rigorous bound; further, there is no guarantee that the error is distributed as a Gaussian, so the error term should be taken only as a rough indication of probable error.

Suppose that you want to integrate a function g over a region W that is not easy to sample randomly. For example, W might have a very complicated shape. No problem. Just find a region V that *includes* W and that *can* easily be sampled (Figure 7.6.1), and then define f to be equal to g for points in W and equal to zero for points outside of W (but still inside the sampled V). You want to try to make V enclose W as closely as possible, because the zero values of f will increase the error estimate term of (7.6.1). And well they should: points chosen outside of W have no information content, so the effective value of N, the number of points, is reduced. The error estimate in (7.6.1) takes this into account.

General purpose routines for Monte Carlo integration are quite complicated (see §7.8), but a worked example will show the underlying simplicity of the method. Suppose that we want to find the weight and the position of the center of mass of an

Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard, 1980, NBS Special Publication 500–20 (Washington: U.S. Department of Commerce, National Bureau of Standards). [3]

Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley). [4]

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 6. [5]

Vitter, J.S., and Chen, W-C. 1987, *Design and Analysis of Coalesced Hashing* (New York: Oxford University Press). [6]

7.6 Simple Monte Carlo Integration

Inspirations for numerical methods can spring from unlikely sources. "Splines" first were flexible strips of wood used by draftsmen. "Simulated annealing" (we shall see in §10.9) is rooted in a thermodynamic analogy. And who does not feel at least a faint echo of glamor in the name "Monte Carlo method"?

Suppose that we pick N random points, uniformly distributed in a multidimensional volume V. Call them $x_1, \ldots, x_N$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f \, dV \approx V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{7.6.1}$$

Here the angle brackets denote taking the arithmetic mean over the N sample points,

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(x_i) \tag{7.6.2}$$

The "plus-or-minus" term in (7.6.1) is a one standard deviation error estimate for the integral, not a rigorous bound; further, there is no guarantee that the error is distributed as a Gaussian, so the error term should be taken only as a rough indication of probable error.

Suppose that you want to integrate a function g over a region W that is not easy to sample randomly. For example, W might have a very complicated shape. No problem. Just find a region V that *includes* W and that *can* easily be sampled (Figure 7.6.1), and then define f to be equal to g for points in W and equal to zero for points outside of W (but still inside the sampled V). You want to try to make V enclose W as closely as possible, because the zero values of f will increase the error estimate term of (7.6.1). And well they should: points chosen outside of W have no information content, so the effective value of N, the number of points, is reduced. The error estimate in (7.6.1) takes this into account.

General purpose routines for Monte Carlo integration are quite complicated (see §7.8), but a worked example will show the underlying simplicity of the method. Suppose that we want to find the weight and the position of the center of mass of an

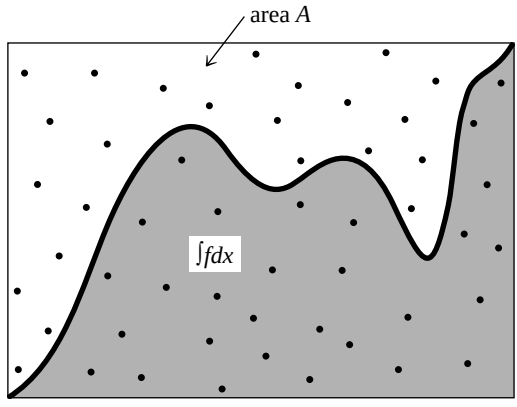


Figure 7.6.1. Monte Carlo integration. Random points are chosen within the area A. The integral of the function f is estimated as the area of A multiplied by the fraction of random points that fall below the curve f. Refinements on this procedure can improve the accuracy of the method; see text.

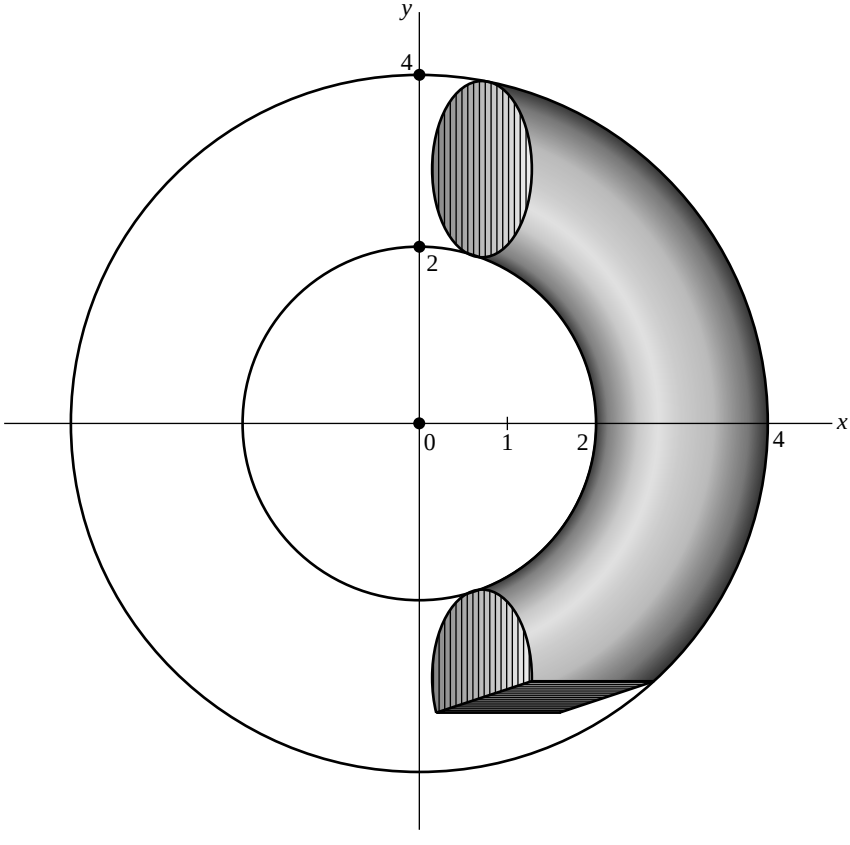


Figure 7.6.2. Example of Monte Carlo integration (see text). The region of interest is a piece of a torus, bounded by the intersection of two planes. The limits of integration of the region cannot easily be written in analytically closed form, so Monte Carlo is a useful technique.

object of complicated shape, namely the intersection of a torus with the edge of a large box. In particular let the object be defined by the three simultaneous conditions

$$z^2 + \left(\sqrt{x^2+y^2} - 3\right)^2 \le 1 \tag{7.6.3}$$

(torus centered on the origin with major radius $= 4$, minor radius $= 2$)

$$x \ge 1 \qquad y \ge -3 \tag{7.6.4}$$

(two faces of the box, see Figure 7.6.2). Suppose for the moment that the object has a constant density ρ.

We want to estimate the following integrals over the interior of the complicated object:

$$\int \rho \, dx\, dy\, dz \qquad \int x\rho \, dx\, dy\, dz \qquad \int y\rho \, dx\, dy\, dz \qquad \int z\rho \, dx\, dy\, dz \tag{7.6.5}$$

The coordinates of the center of mass will be the ratio of the latter three integrals (linear moments) to the first one (the weight).

In the following fragment, the region V, enclosing the piece-of-torus W, is the rectangular box extending from 1 to 4 in x, -3 to 4 in y, and -1 to 1 in z.

```
n=                                      Set to the number of sample points desired.
den=                                    Set to the constant value of the density.
sw=0.                                   Zero the various sums to be accumulated.
swx=0.
swy=0.
swz=0.
varw=0.
varx=0.
vary=0.
varz=0.
vol=3.*7.*2.                            Volume of the sampled region.
do 11 j=1,n
    x=1.+3.*ran2(idum)                  Pick a point randomly in the sampled region.
    y=-3.+7.*ran2(idum)
    z=-1.+2.*ran2(idum)
    if (z**2+(sqrt(x**2+y**2)-3.)**2.le.1.)then      Is it in the torus?
        sw=sw+den                       If so, add to the various cumulants.
        swx=swx+x*den
        swy=swy+y*den
        swz=swz+z*den
        varw=varw+den**2
        varx=varx+(x*den)**2
        vary=vary+(y*den)**2
        varz=varz+(z*den)**2
    endif
enddo 11
w=vol*sw/n                              The values of the integrals (7.6.5),
x=vol*swx/n
y=vol*swy/n
z=vol*swz/n
dw=vol*sqrt((varw/n-(sw/n)**2)/n)       and their corresponding error estimates.
dx=vol*sqrt((varx/n-(swx/n)**2)/n)
dy=vol*sqrt((vary/n-(swy/n)**2)/n)
dz=vol*sqrt((varz/n-(swz/n)**2)/n)
```

A change of variable can often be extremely worthwhile in Monte Carlo integration. Suppose, for example, that we want to evaluate the same integrals, but for a piece-of-torus whose density is a strong function of z, in fact varying according to

$$\rho(x, y, z) = e^{5z} \tag{7.6.6}$$

One way to do this is to put the statement

```
den=exp(5.*z)
```

inside the `if...then` block, just before `den` is first used. This will work, but it is a poor way to proceed. Since (7.6.6) falls so rapidly to zero as z decreases (down to its lower limit -1), most sampled points contribute almost nothing to the sum of the weight or moments. These points are effectively wasted, almost as badly as those that fall outside of the region W. A change of variable, exactly as in the transformation methods of §7.2, solves this problem. Let

$$ds = e^{5z}dz \qquad \text{so that} \qquad s = \frac{1}{5}e^{5z}, \quad z = \frac{1}{5}\ln(5s) \tag{7.6.7}$$

Then $\rho dz = ds$, and the limits $-1 < z < 1$ become $.00135 < s < 29.682$. The program fragment now looks like this

```
n=                                      Set to the number of sample points desired.
sw=0.
swx=0.
swy=0.
swz=0.
varw=0.
varx=0.
vary=0.
varz=0.
ss=(0.2*(exp(5.)-exp(-5.)))             Interval of s to be random sampled.
vol=3.*7.*ss                            Volume in x,y,s-space.
do 11 j=1,n
    x=1.+3.*ran2(idum)
    y=-3.+7.*ran2(idum)
    s=.00135+ss*ran2(idum)              Pick a point in s.
    z=0.2*log(5.*s)                     Equation (7.6.7).
    if (z**2+(sqrt(x**2+y**2)-3.)**2.lt.1.)then
        sw=sw+1.                        Density is 1, since absorbed into definition of s.
        swx=swx+x
        swy=swy+y
        swz=swz+z
        varw=varw+1.
        varx=varx+x**2
        vary=vary+y**2
        varz=varz+z**2
    endif
enddo 11
w=vol*sw/n                              The values of the integrals (7.6.5),
x=vol*swx/n
y=vol*swy/n
z=vol*swz/n
dw=vol*sqrt((varw/n-(sw/n)**2)/n)       and their corresponding error estimates.
dx=vol*sqrt((varx/n-(swx/n)**2)/n)
dy=vol*sqrt((vary/n-(swy/n)**2)/n)
dz=vol*sqrt((varz/n-(swz/n)**2)/n)
```

If you think for a minute, you will realize that equation (7.6.7) was useful only because the part of the integrand that we wanted to eliminate (e^{5z}) was both integrable analytically, and had an integral that could be analytically inverted. (Compare §7.2.) In general these properties will not hold. Question: What then? Answer: Pull out of the integrand the "best" factor that *can* be integrated and inverted. The criterion for "best" is to try to reduce the remaining integrand to a function that is as close as possible to constant.

The limiting case is instructive: If you manage to make the integrand f *exactly* constant, and if the region V, of known volume, *exactly* encloses the desired region W, then the average of f that you compute will be exactly its constant value, and the error estimate in equation (7.6.1) will exactly vanish. You will, in fact, have done the integral exactly, and the Monte Carlo numerical evaluations are superfluous. So, backing off from the extreme limiting case, *to the extent* that you are able to make f approximately constant by change of variable, and *to the extent* that you can sample a region only slightly larger than W, you will increase the accuracy of the Monte Carlo integral. This technique is generically called *reduction of variance* in the literature.

The fundamental disadvantage of simple Monte Carlo integration is that its accuracy increases only as the square root of N, the number of sampled points. If your accuracy requirements are modest, or if your computer budget is large, then the technique is highly recommended as one of great generality. In the next two sections we will see that there are techniques available for "breaking the square root of N barrier" and achieving, at least in some cases, higher accuracy with fewer function evaluations.

CITED REFERENCES AND FURTHER READING:

Hammersley, J.M., and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).

Shreider, Yu. A. (ed.) 1966, *The Monte Carlo Method* (Oxford: Pergamon).

Sobol', I.M. 1974, *The Monte Carlo Method* (Chicago: University of Chicago Press).

Kalos, M.H., and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

7.7 Quasi- (that is, Sub-) Random Sequences

We have just seen that choosing N points uniformly randomly in an n-dimensional space leads to an error term in Monte Carlo integration that decreases as $1/\sqrt{N}$. In essence, each new point sampled adds linearly to an accumulated sum that will become the function average, and also linearly to an accumulated sum of squares that will become the variance (equation 7.6.2). The estimated error comes from the square root of this variance, hence the power $N^{-1/2}$.

Just because this square root convergence is familiar does not, however, mean that it is inevitable. A simple counterexample is to choose sample points that lie on a Cartesian grid, and to sample each grid point exactly once (in whatever order). The Monte Carlo method thus becomes a deterministic quadrature scheme — albeit a simple one — whose fractional error decreases at least as fast as N^{-1} (even faster if the function goes to zero smoothly at the boundaries of the sampled region, or is periodic in the region).

If you think for a minute, you will realize that equation (7.6.7) was useful only because the part of the integrand that we wanted to eliminate (e^{5z}) was both integrable analytically, and had an integral that could be analytically inverted. (Compare §7.2.) In general these properties will not hold. Question: What then? Answer: Pull out of the integrand the "best" factor that *can* be integrated and inverted. The criterion for "best" is to try to reduce the remaining integrand to a function that is as close as possible to constant.

The limiting case is instructive: If you manage to make the integrand f *exactly* constant, and if the region V, of known volume, *exactly* encloses the desired region W, then the average of f that you compute will be exactly its constant value, and the error estimate in equation (7.6.1) will exactly vanish. You will, in fact, have done the integral exactly, and the Monte Carlo numerical evaluations are superfluous. So, backing off from the extreme limiting case, *to the extent* that you are able to make f approximately constant by change of variable, and *to the extent* that you can sample a region only slightly larger than W, you will increase the accuracy of the Monte Carlo integral. This technique is generically called *reduction of variance* in the literature.

The fundamental disadvantage of simple Monte Carlo integration is that its accuracy increases only as the square root of N, the number of sampled points. If your accuracy requirements are modest, or if your computer budget is large, then the technique is highly recommended as one of great generality. In the next two sections we will see that there are techniques available for "breaking the square root of N barrier" and achieving, at least in some cases, higher accuracy with fewer function evaluations.

CITED REFERENCES AND FURTHER READING:

Hammersley, J.M., and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).

Shreider, Yu. A. (ed.) 1966, *The Monte Carlo Method* (Oxford: Pergamon).

Sobol', I.M. 1974, *The Monte Carlo Method* (Chicago: University of Chicago Press).

Kalos, M.H., and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

7.7 Quasi- (that is, Sub-) Random Sequences

We have just seen that choosing N points uniformly randomly in an n-dimensional space leads to an error term in Monte Carlo integration that decreases as $1/\sqrt{N}$. In essence, each new point sampled adds linearly to an accumulated sum that will become the function average, and also linearly to an accumulated sum of squares that will become the variance (equation 7.6.2). The estimated error comes from the square root of this variance, hence the power $N^{-1/2}$.

Just because this square root convergence is familiar does not, however, mean that it is inevitable. A simple counterexample is to choose sample points that lie on a Cartesian grid, and to sample each grid point exactly once (in whatever order). The Monte Carlo method thus becomes a deterministic quadrature scheme — albeit a simple one — whose fractional error decreases at least as fast as N^{-1} (even faster if the function goes to zero smoothly at the boundaries of the sampled region, or is periodic in the region).

The trouble with a grid is that one has to decide *in advance* how fine it should be. One is then committed to completing all of its sample points. With a grid, it is not convenient to "sample *until*" some convergence or termination criterion is met. One might ask if there is not some intermediate scheme, some way to pick sample points "at random," yet spread out in some self-avoiding way, avoiding the chance clustering that occurs with uniformly random points.

A similar question arises for tasks other than Monte Carlo integration. We might want to search an n-dimensional space for a point where some (locally computable) condition holds. Of course, for the task to be computationally meaningful, there had better be continuity, so that the desired condition will hold in some finite n-dimensional neighborhood. We may not know *a priori* how large that neighborhood is, however. We want to "sample *until*" the desired point is found, moving smoothly to finer scales with increasing samples. Is there any way to do this that is better than uncorrelated, random samples?

The answer to the above question is "yes." Sequences of n-tuples that fill n-space more uniformly than uncorrelated random points are called *quasi-random sequences*. That term is somewhat of a misnomer, since there is nothing "random" about quasi-random sequences: They are cleverly crafted to be, in fact, *sub*-random. The sample points in a quasi-random sequence are, in a precise sense, "maximally avoiding" of each other.

A conceptually simple example is *Halton's sequence* [1]. In one dimension, the jth number H_j in the sequence is obtained by the following steps: (i) Write j as a number in base b, where b is some prime. (For example $j = 17$ in base $b = 3$ is 122.) (ii) Reverse the digits and put a radix point (i.e., a decimal point base b) in front of the sequence. (In the example, we get 0.221 base 3.) The result is H_j. To get a sequence of n-tuples in n-space, you make each component a Halton sequence with a different prime base b. Typically, the first n primes are used.

It is not hard to see how Halton's sequence works: Every time the number of digits in j increases by one place, j's digit-reversed fraction becomes a factor of b finer-meshed. Thus the process is one of filling in all the points on a sequence of finer and finer Cartesian grids — and in a kind of maximally spread-out order on each grid (since, e.g., the most rapidly changing digit in j controls the *most* significant digit of the fraction).

Other ways of generating quasi-random sequences have been suggested by Faure, Sobol', Niederreiter, and others. Bratley and Fox [2] provide a good review and references, and discuss a particularly efficient variant of the Sobol' [3] sequence suggested by Antonov and Saleev [4]. It is this Antonov-Saleev variant whose implementation we now discuss.

The Sobol' sequence generates numbers between zero and one directly as binary fractions of length w bits, from a set of w special binary fractions, V_i, $i = 1, 2, \ldots, w$, called *direction numbers*. In Sobol's original method, the jth number X_j is generated by XORing (bitwise exclusive or) together the set of V_i's satisfying the criterion on i, "the ith bit of j is nonzero." As j increments, in other words, different ones of the V_i's flash in and out of X_j on different time scales. V_1 alternates between being present and absent most quickly, while V_k goes from present to absent (or vice versa) only every 2^{k-1} steps.

Antonov and Saleev's contribution was to show that instead of using the bits of the integer j to select direction numbers, one could just as well use the bits of the *Gray code* of j, $G(j)$. (For a quick review of Gray codes, look at §20.2.)

Now $G(j)$ and $G(j+1)$ differ in exactly one bit position, namely in the position of the

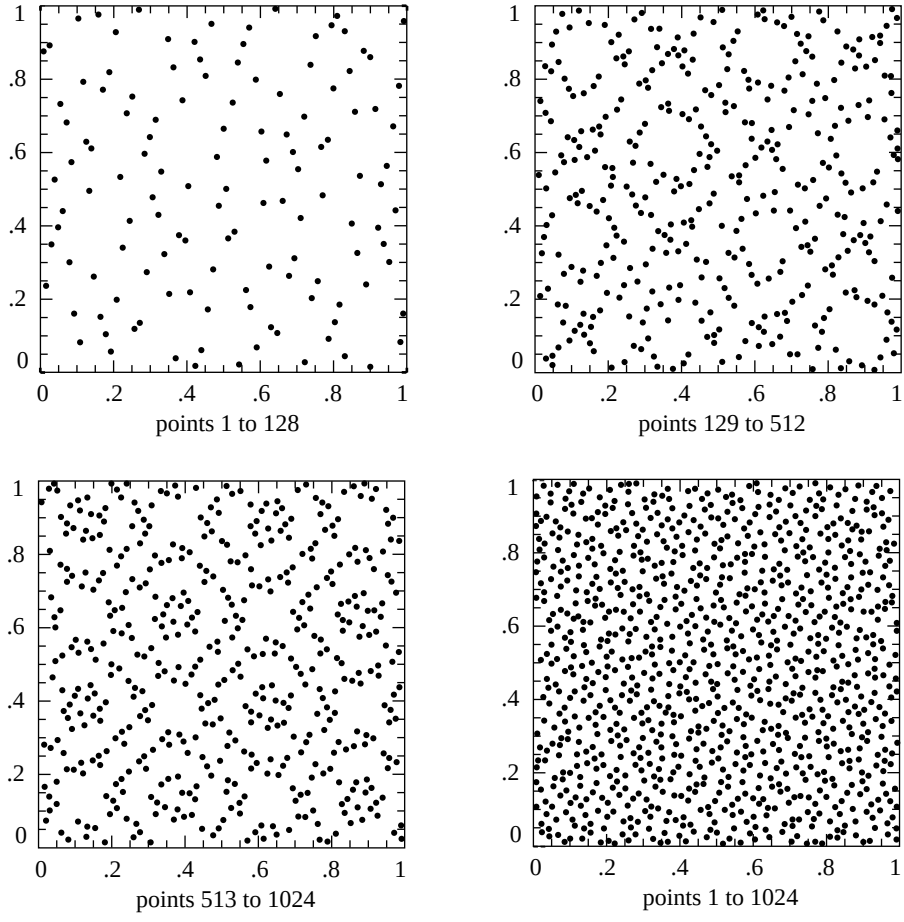


Figure 7.7.1. First 1024 points of a two-dimensional Sobol' sequence. The sequence is generated number-theoretically, rather than randomly, so successive points at any stage "know" how to fill in the gaps in the previously generated distribution.

rightmost zero bit in the binary representation of j (adding a leading zero to j if necessary). A consequence is that the $j+1$st Sobol'-Antonov-Saleev number can be obtained from the jth by XORing it with *a single* V_i, namely with i the position of the rightmost zero bit in j. This makes the calculation of the sequence very efficient, as we shall see.

Figure 7.7.1 plots the first 1024 points generated by a two-dimensional Sobol' sequence. One sees that successive points do "know" about the gaps left previously, and keep filling them in, hierarchically.

We have deferred to this point a discussion of how the direction numbers V_i are generated. Some nontrivial mathematics is involved in that, so we will content ourself with a cookbook summary only: Each different Sobol' sequence (or component of an n-dimensional sequence) is based on a different primitive polynomial over the integers modulo 2, that is, a polynomial whose coefficients are either 0 or 1, and which generates a maximal length shift register sequence. (Primitive polynomials modulo 2 were used in §7.4, and are further discussed in §20.3.) Suppose P is such a polynomial, of degree q,

$$P = x^q + a_1 x^{q-1} + a_2 x^{q-2} + \cdots + a_{q-1} + 1 \tag{7.7.1}$$

Degree	Primitive Polynomials Modulo 2*
1	0 (i.e., $x + 1$)
2	1 (i.e., $x^2 + x + 1$)
3	1, 2 (i.e., $x^3 + x + 1$ and $x^3 + x^2 + 1$)
4	1, 4 (i.e., $x^4 + x + 1$ and $x^4 + x^3 + 1$)
5	2, 4, 7, 11, 13, 14
6	1, 13, 16, 19, 22, 25
7	1, 4, 7, 8, 14, 19, 21, 28, 31, 32, 37, 41, 42, 50, 55, 56, 59, 62
8	14, 21, 22, 38, 47, 49, 50, 52, 56, 67, 70, 84, 97, 103, 115, 122
9	8, 13, 16, 22, 25, 44, 47, 52, 55, 59, 62, 67, 74, 81, 82, 87, 91, 94, 103, 104, 109, 122, 124, 137, 138, 143, 145, 152, 157, 167, 173, 176, 181, 182, 185, 191, 194, 199, 218, 220, 227, 229, 230, 234, 236, 241, 244, 253
10	4, 13, 19, 22, 50, 55, 64, 69, 98, 107, 115, 121, 127, 134, 140, 145, 152, 158, 161, 171, 181, 194, 199, 203, 208, 227, 242, 251, 253, 265, 266, 274, 283, 289, 295, 301, 316, 319, 324, 346, 352, 361, 367, 382, 395, 398, 400, 412, 419, 422, 426, 428, 433, 446, 454, 457, 472, 493, 505, 508

*Expressed as a decimal integer representing the interior bits (that is, omitting the high-order bit and the unit bit).

Define a sequence of integers M_i by the q-term recurrence relation,

$$M_i = 2a_1 M_{i-1} \oplus 2^2 a_2 M_{i-2} \oplus \cdots \oplus 2^{q-1} M_{i-q+1} a_{q-1} \oplus (2^q M_{i-q} \oplus M_{i-q}) \quad (7.7.2)$$

Here bitwise XOR is denoted by $\oplus$. The starting values for this recurrence are that $M_1, \ldots, M_q$ can be arbitrary odd integers less than $2, \ldots, 2^q$, respectively. Then, the direction numbers V_i are given by

$$V_i = M_i / 2^i \qquad i = 1, \ldots, w \quad (7.7.3)$$

The accompanying table lists all primitive polynomials modulo 2 with degree $q \leq 10$. Since the coefficients are either 0 or 1, and since the coefficients of x^q and of 1 are predictably 1, it is convenient to denote a polynomial by its middle coefficients taken as the bits of a binary number (higher powers of x being more significant bits). The table uses this convention.

Turn now to the implementation of the Sobol' sequence. Successive calls to the function `sobseq` (after a preliminary initializing call) return successive points in an n-dimensional Sobol' sequence based on the first n primitive polynomials in the table. As given, the routine is initialized for maximum n of 6 dimensions, and for a word length w of 30 bits. These parameters can be altered by changing `MAXBIT` ($\equiv w$) and `MAXDIM`, and by adding more initializing data to the arrays `ip` (the primitive polynomials from the table), `mdeg` (their degrees), and `iv` (the starting values for the recurrence, equation 7.7.2). A second table, below, elucidates the initializing data in the routine.

```
SUBROUTINE sobseq(n,x)
INTEGER n,MAXBIT,MAXDIM
REAL x(*)
PARAMETER (MAXBIT=30,MAXDIM=6)
```

When `n` is negative, internally initializes a set of `MAXBIT` direction numbers for each of `MAXDIM` different Sobol' sequences. When `n` is positive (but $\leq$`MAXDIM`), returns as the

Initializing Values Used in sobseq					
Degree	Polynomial	Starting Values			
1	0	1	(3)	(5)	(15) ...
2	1	1	1	(7)	(11) ...
3	1	1	3	7	(5) ...
3	2	1	3	3	(15) ...
4	1	1	1	3	13 ...
4	4	1	1	5	9 ...

Parenthesized values are not freely specifiable, but are forced by the required recurrence for this degree.

vector x(1..n) the next values from n of these sequences. (n must not be changed between initializations.)

```
      INTEGER i,im,in,ipp,j,k,l,ip(MAXDIM),iu(MAXDIM,MAXBIT),
     *     iv(MAXBIT*MAXDIM),ix(MAXDIM),mdeg(MAXDIM)
      REAL fac
      SAVE ip,mdeg,ix,iv,in,fac
      EQUIVALENCE (iv,iu)                   To allow both 1D and 2D addressing.
      DATA ip /0,1,1,2,1,4/, mdeg /1,2,3,3,4,4/, ix /6*0/
      DATA iv /6*1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9,156*0/
      if (n.lt.0) then                      Initialize, don't return a vector.
        do 11 k=1,MAXDIM
          ix(k)=0
        enddo 11
        in=0
        if(iv(1).ne.1)return
        fac=1./2.**MAXBIT
        do 15 k=1,MAXDIM
          do 12 j=1,mdeg(k)                 Stored values only require normalization.
            iu(k,j)=iu(k,j)*2**(MAXBIT-j)
          enddo 12
          do 14 j=mdeg(k)+1,MAXBIT          Use the recurrence to get other values.
            ipp=ip(k)
            i=iu(k,j-mdeg(k))
            i=ieor(i,i/2**mdeg(k))
            do 13 l=mdeg(k)-1,1,-1
              if(iand(ipp,1).ne.0)i=ieor(i,iu(k,j-l))
              ipp=ipp/2
            enddo 13
            iu(k,j)=i
          enddo 14
        enddo 15
      else                                  Calculate the next vector in the sequence.
        im=in
        do 16 j=1,MAXBIT                    Find the rightmost zero bit.
          if(iand(im,1).eq.0)goto 1
          im=im/2
        enddo 16
        pause 'MAXBIT too small in sobseq'
1       im=(j-1)*MAXDIM
        do 17 k=1,min(n,MAXDIM)             XOR the appropriate direction number into each com-
          ix(k)=ieor(ix(k),iv(im+k))          ponent of the vector and convert to a floating
          x(k)=ix(k)*fac                      number.
        enddo 17
        in=in+1                             Increment the counter.
```

```
endif
return
END
```

How good is a Sobol' sequence, anyway? For Monte Carlo integration of a smooth function in n dimensions, the answer is that the fractional error will decrease with N, the number of samples, as $(\ln N)^n/N$, i.e., almost as fast as $1/N$. As an example, let us integrate a function that is nonzero inside a torus (doughnut) in three-dimensional space. If the major radius of the torus is R_0, the minor radial coordinate r is defined by

$$r = \left([(x^2 + y^2)^{1/2} - R_0]^2 + z^2 \right)^{1/2} \tag{7.7.4}$$

Let us try the function

$$f(x,y,z) = \begin{cases} 1 + \cos\left(\dfrac{\pi r^2}{a^2}\right) & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.7.5}$$

which can be integrated analytically in cylindrical coordinates, giving

$$\int\int\int dx\, dy\, dz\, f(x,y,z) = 2\pi^2 a^2 R_0 \tag{7.7.6}$$

With parameters $R_0 = 0.6$, $r_0 = 0.3$, we did 100 successive Monte Carlo integrations of equation (7.7.4), sampling uniformly in the region $-1 < x, y, z < 1$, for the two cases of uncorrelated random points and the Sobol' sequence generated by the routine `sobseq`. Figure 7.7.2 shows the results, plotting the r.m.s. average error of the 100 integrations as a function of the number of points sampled. (For any *single* integration, the error of course wanders from positive to negative, or vice versa, so a logarithmic plot of fractional error is not very informative.) The thin, dashed curve corresponds to uncorrelated random points and shows the familiar $N^{-1/2}$ asymptotics. The thin, solid gray curve shows the result for the Sobol' sequence. The logarithmic term in the expected $(\ln N)^3/N$ is readily apparent as curvature in the curve, but the asymptotic N^{-1} is unmistakable.

To understand the importance of Figure 7.7.2, suppose that a Monte Carlo integration of f with 1% accuracy is desired. The Sobol' sequence achieves this accuracy in a few thousand samples, while pseudorandom sampling requires nearly 100,000 samples. The ratio would be even greater for higher desired accuracies.

A different, not quite so favorable, case occurs when the function being integrated has hard (discontinuous) boundaries inside the sampling region, for example the function that is one inside the torus, zero outside,

$$f(x,y,z) = \begin{cases} 1 & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.7.7}$$

where r is defined in equation (7.7.4). Not by coincidence, this function has the same analytic integral as the function of equation (7.7.5), namely $2\pi^2 a^2 R_0$.

The carefully hierarchical Sobol' sequence is based on a set of Cartesian grids, but the boundary of the torus has no particular relation to those grids. The result is that it is essentially random whether sampled points in a thin layer at the surface of the torus, containing on the order of $N^{2/3}$ points, come out to be inside, or outside, the torus. The square root law, applied to this thin layer, gives $N^{1/3}$ fluctuations in the sum, or $N^{-2/3}$ fractional error in the Monte Carlo integral. One sees this behavior verified in Figure 7.7.2 by the thicker gray curve. The thicker dashed curve in Figure 7.7.2 is the result of integrating the function of equation (7.7.7) using independent random points. While the advantage of the Sobol' sequence is not quite so dramatic as in the case of a smooth function, it can nonetheless be a significant factor ($\sim$5) even at modest accuracies like 1%, and greater at higher accuracies.

Note that we have not provided the routine `sobseq` with a means of starting the sequence at a point other than the beginning, but this feature would be easy to add. Once the initialization of the direction numbers `iv` has been done, the jth point can be obtained directly by XORing together those direction numbers corresponding to nonzero bits in the Gray code of j, as described above.

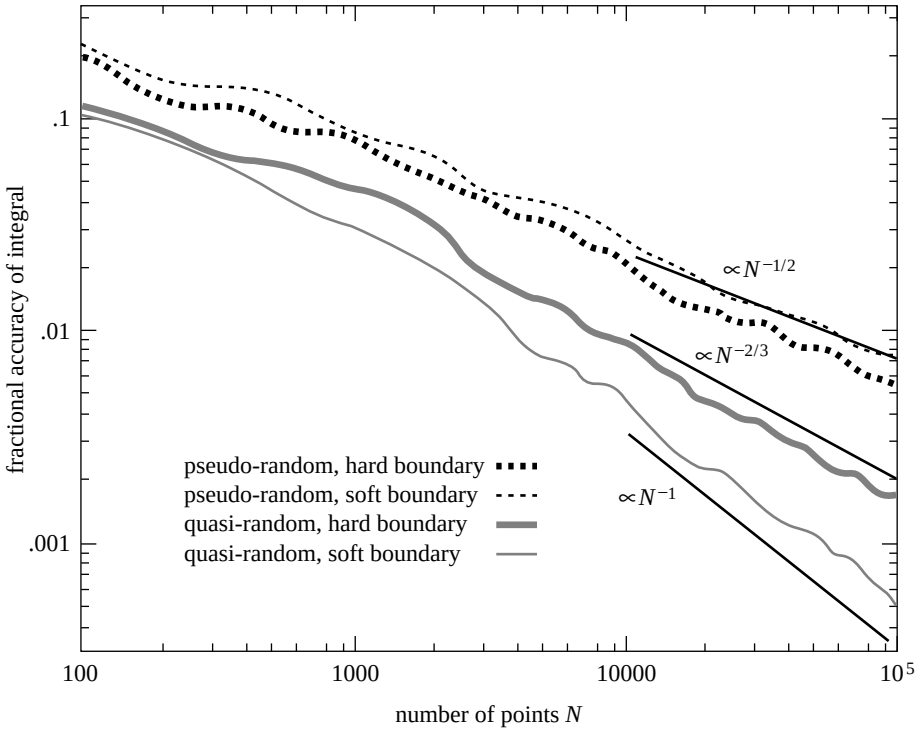


Figure 7.7.2. Fractional accuracy of Monte Carlo integrations as a function of number of points sampled, for two different integrands and two different methods of choosing random points. The quasi-random Sobol' sequence converges much more rapidly than a conventional pseudo-random sequence. Quasi-random sampling does better when the integrand is smooth ("soft boundary") than when it has step discontinuities ("hard boundary"). The curves shown are the r.m.s. average of 100 trials.

The Latin Hypercube

We might here give passing mention the unrelated technique of *Latin square* or *Latin hypercube* sampling, which is useful when you must sample an N-dimensional space *exceedingly* sparsely, at M points. For example, you may want to test the crashworthiness of cars as a simultaneous function of 4 different design parameters, but with a budget of only three expendable cars. (The issue is not whether this is a good plan — it isn't — but rather how to make the best of the situation!)

The idea is to partition each design parameter (dimension) into M segments, so that the whole space is partitioned into M^N cells. (You can choose the segments in each dimension to be equal or unequal, according to taste.) With 4 parameters and 3 cars, for example, you end up with $3 \times 3 \times 3 \times 3 = 81$ cells.

Next, choose M cells to contain the sample points by the following algorithm: Randomly choose one of the M^N cells for the first point. Now eliminate all cells that agree with this point on *any* of its parameters (that is, cross out all cells in the same row, column, etc.), leaving $(M-1)^N$ candidates. Randomly choose one of these, eliminate new rows and columns, and continue the process until there is only one cell left, which then contains the final sample point.

The result of this construction is that *each* design parameter will have been tested in *every one* of its subranges. If the response of the system under test is

dominated by *one* of the design parameters, that parameter will be found with this sampling technique. On the other hand, if there is an important interaction among different design parameters, then the Latin hypercube gives no particular advantage. Use with care.

CITED REFERENCES AND FURTHER READING:

Halton, J.H. 1960, *Numerische Mathematik*, vol. 2, pp. 84–90. [1]

Bratley P., and Fox, B.L. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 88–100. [2]

Lambert, J.P. 1988, in *Numerical Mathematics – Singapore 1988*, ISNM vol. 86, R.P. Agarwal, Y.M. Chow, and S.J. Wilson, eds. (Basel: Birkhaüser), pp. 273–284.

Niederreiter, H. 1988, in *Numerical Integration III*, ISNM vol. 85, H. Brass and G. Hämmerlin, eds. (Basel: Birkhaüser), pp. 157–171.

Sobol', I.M. 1967, *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112. [3]

Antonov, I.A., and Saleev, V.M 1979, *USSR Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252–256. [4]

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York, Wiley) [discusses Latin Square].

7.8 Adaptive and Recursive Monte Carlo Methods

This section discusses more advanced techniques of Monte Carlo integration. As examples of the use of these techniques, we include two rather different, fairly sophisticated, multidimensional Monte Carlo codes: `vegas` [1,2], and `miser` [3]. The techniques that we discuss all fall under the general rubric of *reduction of variance* (§7.6), but are otherwise quite distinct.

Importance Sampling

The use of *importance sampling* was already implicit in equations (7.6.6) and (7.6.7). We now return to it in a slightly more formal way. Suppose that an integrand f can be written as the product of a function h that is almost constant times another, positive, function g. Then its integral over a multidimensional volume V is

$$\int f\,dV = \int (f/g)\,g dV = \int h\,g dV \tag{7.8.1}$$

In equation (7.6.7) we interpreted equation (7.8.1) as suggesting a change of variable to G, the indefinite integral of g. That made gdV a perfect differential. We then proceeded to use the basic theorem of Monte Carlo integration, equation (7.6.1). A more general interpretation of equation (7.8.1) is that we can integrate f by instead sampling h — not, however, with uniform probability density dV, but rather with nonuniform density gdV. In this second interpretation, the first interpretation follows as the special case, where the *means* of generating the nonuniform sampling of gdV is via the transformation method, using the indefinite integral G (see §7.2).

More directly, one can go back and generalize the basic theorem (7.6.1) to the case of nonuniform sampling: Suppose that points x_i are chosen within the volume V with a probability density p satisfying

$$\int p\,dV = 1 \tag{7.8.2}$$

dominated by *one* of the design parameters, that parameter will be found with this sampling technique. On the other hand, if there is an important interaction among different design parameters, then the Latin hypercube gives no particular advantage. Use with care.

CITED REFERENCES AND FURTHER READING:

Halton, J.H. 1960, *Numerische Mathematik*, vol. 2, pp. 84–90. [1]

Bratley P., and Fox, B.L. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 88–100. [2]

Lambert, J.P. 1988, in *Numerical Mathematics – Singapore 1988*, ISNM vol. 86, R.P. Agarwal, Y.M. Chow, and S.J. Wilson, eds. (Basel: Birkhaüser), pp. 273–284.

Niederreiter, H. 1988, in *Numerical Integration III*, ISNM vol. 85, H. Brass and G. Hämmerlin, eds. (Basel: Birkhaüser), pp. 157–171.

Sobol', I.M. 1967, *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112. [3]

Antonov, I.A., and Saleev, V.M 1979, *USSR Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252–256. [4]

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York, Wiley) [discusses Latin Square].

7.8 Adaptive and Recursive Monte Carlo Methods

This section discusses more advanced techniques of Monte Carlo integration. As examples of the use of these techniques, we include two rather different, fairly sophisticated, multidimensional Monte Carlo codes: `vegas` [1,2], and `miser` [3]. The techniques that we discuss all fall under the general rubric of *reduction of variance* (§7.6), but are otherwise quite distinct.

Importance Sampling

The use of *importance sampling* was already implicit in equations (7.6.6) and (7.6.7). We now return to it in a slightly more formal way. Suppose that an integrand f can be written as the product of a function h that is almost constant times another, positive, function g. Then its integral over a multidimensional volume V is

$$\int f\,dV = \int (f/g)\,g dV = \int h\,g dV \tag{7.8.1}$$

In equation (7.6.7) we interpreted equation (7.8.1) as suggesting a change of variable to G, the indefinite integral of g. That made gdV a perfect differential. We then proceeded to use the basic theorem of Monte Carlo integration, equation (7.6.1). A more general interpretation of equation (7.8.1) is that we can integrate f by instead sampling h — not, however, with uniform probability density dV, but rather with nonuniform density gdV. In this second interpretation, the first interpretation follows as the special case, where the *means* of generating the nonuniform sampling of gdV is via the transformation method, using the indefinite integral G (see §7.2).

More directly, one can go back and generalize the basic theorem (7.6.1) to the case of nonuniform sampling: Suppose that points x_i are chosen within the volume V with a probability density p satisfying

$$\int p\,dV = 1 \tag{7.8.2}$$

The generalized fundamental theorem is that the integral of any function f is estimated, using N sample points $x_i, \ldots, x_N$, by

$$I \equiv \int f\, dV = \int \frac{f}{p}\, p dV \approx \left\langle \frac{f}{p} \right\rangle \pm \sqrt{\frac{\langle f^2/p^2 \rangle - \langle f/p \rangle^2}{N}} \tag{7.8.3}$$

where angle brackets denote arithmetic means over the N points, exactly as in equation (7.6.2). As in equation (7.6.1), the "plus-or-minus" term is a one standard deviation error estimate. Notice that equation (7.6.1) is in fact the special case of equation (7.8.3), with $p = \text{constant} = 1/V$.

What is the best choice for the sampling density p? Intuitively, we have already seen that the idea is to make $h = f/p$ as close to constant as possible. We can be more rigorous by focusing on the numerator inside the square root in equation (7.8.3), which is the variance per sample point. Both angle brackets are themselves Monte Carlo estimators of integrals, so we can write

$$S \equiv \left\langle \frac{f^2}{p^2} \right\rangle - \left\langle \frac{f}{p} \right\rangle^2 \approx \int \frac{f^2}{p^2}\, p dV - \left[\int \frac{f}{p}\, p dV \right]^2 = \int \frac{f^2}{p}\, dV - \left[\int f\, dV \right]^2 \tag{7.8.4}$$

We now find the optimal p subject to the constraint equation (7.8.2) by the functional variation

$$0 = \frac{\delta}{\delta p} \left(\int \frac{f^2}{p}\, dV - \left[\int f\, dV \right]^2 + \lambda \int p\, dV \right) \tag{7.8.5}$$

with λ a Lagrange multiplier. Note that the middle term does not depend on p. The variation (which comes inside the integrals) gives $0 = -f^2/p^2 + \lambda$ or

$$p = \frac{|f|}{\sqrt{\lambda}} = \frac{|f|}{\int |f|\, dV} \tag{7.8.6}$$

where λ has been chosen to enforce the constraint (7.8.2).

If f has one sign in the region of integration, then we get the obvious result that the optimal choice of p — if one can figure out a practical way of effecting the sampling — is that it be proportional to $|f|$. Then the variance is reduced to zero. Not so obvious, but seen to be true, is the fact that $p \propto |f|$ is optimal even if f takes on both signs. In that case the variance per sample point (from equations 7.8.4 and 7.8.6) is

$$S = S_{\text{optimal}} = \left(\int |f|\, dV \right)^2 - \left(\int f\, dV \right)^2 \tag{7.8.7}$$

One curiosity is that one can add a constant to the integrand to make it all of one sign, since this changes the integral by a known amount, constant $\times V$. Then, the optimal choice of p always gives zero variance, that is, a perfectly accurate integral! The resolution of this seeming paradox (already mentioned at the end of §7.6) is that perfect knowledge of p in equation (7.8.6) requires perfect knowledge of $\int |f| dV$, which is tantamount to already knowing the integral you are trying to compute!

If your function f takes on a known constant value in most of the volume V, it is certainly a good idea to add a constant so as to make that value zero. Having done that, the accuracy attainable by importance sampling depends in practice not on how small equation (7.8.7) is, but rather on how small is equation (7.8.4) for an *implementable* p, likely only a crude approximation to the ideal.

Stratified Sampling

The idea of *stratified sampling* is quite different from importance sampling. Let us expand our notation slightly and let $\langle\langle f \rangle\rangle$ denote the true average of the function f over the volume V (namely the integral divided by V), while $\langle f \rangle$ denotes as before the simplest (uniformly sampled) Monte Carlo *estimator* of that average:

$$\langle\langle f \rangle\rangle \equiv \frac{1}{V}\int f\, dV \qquad \langle f \rangle \equiv \frac{1}{N}\sum_i f(x_i) \tag{7.8.8}$$

The variance of the estimator, $\mathrm{Var}\left(\langle f \rangle\right)$, which measures the square of the error of the Monte Carlo integration, is asymptotically related to the variance of the function, $\mathrm{Var}\left(f\right) \equiv \langle\langle f^2 \rangle\rangle - \langle\langle f \rangle\rangle^2$, by the relation

$$\mathrm{Var}\left(\langle f \rangle\right) = \frac{\mathrm{Var}\left(f\right)}{N} \tag{7.8.9}$$

(compare equation 7.6.1).

Suppose we divide the volume V into two equal, disjoint subvolumes, denoted a and b, and sample $N/2$ points in each subvolume. Then another estimator for $\langle\langle f \rangle\rangle$, different from equation (7.8.8), which we denote $\langle f \rangle'$, is

$$\langle f \rangle' \equiv \frac{1}{2}\left(\langle f \rangle_a + \langle f \rangle_b\right) \tag{7.8.10}$$

in other words, the mean of the sample averages in the two half-regions. The variance of estimator (7.8.10) is given by

$$\begin{aligned}\mathrm{Var}\left(\langle f \rangle'\right) &= \frac{1}{4}\left[\mathrm{Var}\left(\langle f \rangle_a\right) + \mathrm{Var}\left(\langle f \rangle_b\right)\right] \\ &= \frac{1}{4}\left[\frac{\mathrm{Var}_a\left(f\right)}{N/2} + \frac{\mathrm{Var}_b\left(f\right)}{N/2}\right] \\ &= \frac{1}{2N}\left[\mathrm{Var}_a\left(f\right) + \mathrm{Var}_b\left(f\right)\right]\end{aligned} \tag{7.8.11}$$

Here $\mathrm{Var}_a\left(f\right)$ denotes the variance of f in subregion a, that is, $\langle\langle f^2 \rangle\rangle_a - \langle\langle f \rangle\rangle_a^2$, and correspondingly for b.

From the definitions already given, it is not difficult to prove the relation

$$\mathrm{Var}\left(f\right) = \frac{1}{2}\left[\mathrm{Var}_a\left(f\right) + \mathrm{Var}_b\left(f\right)\right] + \frac{1}{4}\left(\langle\langle f \rangle\rangle_a - \langle\langle f \rangle\rangle_b\right)^2 \tag{7.8.12}$$

(In physics, this formula for combining second moments is the "parallel axis theorem.") Comparing equations (7.8.9), (7.8.11), and (7.8.12), one sees that the stratified (into two subvolumes) sampling gives a variance that is never larger than the simple Monte Carlo case — and smaller whenever the means of the stratified samples, $\langle\langle f \rangle\rangle_a$ and $\langle\langle f \rangle\rangle_b$, are different.

We have not yet exploited the possibility of sampling the two subvolumes with *different numbers* of points, say N_a in subregion a and $N_b \equiv N - N_a$ in subregion b. Let us do so now. Then the variance of the estimator is

$$\mathrm{Var}\left(\langle f \rangle'\right) = \frac{1}{4}\left[\frac{\mathrm{Var}_a\left(f\right)}{N_a} + \frac{\mathrm{Var}_b\left(f\right)}{N - N_a}\right] \tag{7.8.13}$$

which is minimized (one can easily verify) when

$$\frac{N_a}{N} = \frac{\sigma_a}{\sigma_a + \sigma_b} \tag{7.8.14}$$

Here we have adopted the shorthand notation $\sigma_a \equiv \left[\mathrm{Var}_a\left(f\right)\right]^{1/2}$, and correspondingly for b. If N_a satisfies equation (7.8.14), then equation (7.8.13) reduces to

$$\mathrm{Var}\left(\langle f \rangle'\right) = \frac{\left(\sigma_a + \sigma_b\right)^2}{4N} \tag{7.8.15}$$

Equation (7.8.15) reduces to equation (7.8.9) if $\mathrm{Var}\,(f) = \mathrm{Var}_a\,(f) = \mathrm{Var}_b\,(f)$, in which case stratifying the sample makes no difference.

A standard way to generalize the above result is to consider the volume V divided into more than two equal subregions. One can readily obtain the result that the optimal allocation of sample points among the regions is to have the number of points in each region j proportional to σ_j (that is, the square root of the variance of the function f in that subregion). In spaces of high dimensionality (say $d \gtrsim 4$) this is not in practice very useful, however. Dividing a volume into K segments along each dimension implies K^d subvolumes, typically much too large a number when one contemplates estimating all the corresponding σ_j's.

Mixed Strategies

Importance sampling and stratified sampling seem, at first sight, inconsistent with each other. The former concentrates sample points where the magnitude of the integrand $|f|$ is largest, that latter where the variance of f is largest. How can both be right?

The answer is that (like so much else in life) it all depends on what you know and how well you know it. Importance sampling depends on already knowing some approximation to your integral, so that you are able to generate random points x_i with the desired probability density p. To the extent that your p is not ideal, you are left with an error that decreases only as $N^{-1/2}$. Things are particularly bad if your p is far from ideal in a region where the integrand f is changing rapidly, since then the sampled function $h = f/p$ will have a large variance. Importance sampling works by smoothing the values of the sampled function h, and is effective only to the extent that you succeed in this.

Stratified sampling, by contrast, does not necessarily require that you know anything about f. Stratified sampling works by smoothing out the fluctuations of the *number* of points in subregions, not by smoothing the values of the points. The simplest stratified strategy, dividing V into N equal subregions and choosing one point randomly in each subregion, already gives a method whose error decreases asymptotically as N^{-1}, much faster than $N^{-1/2}$. (Note that quasi-random numbers, §7.7, are another way of smoothing fluctuations in the density of points, giving nearly as good a result as the "blind" stratification strategy.)

However, "asymptotically" is an important caveat: For example, if the integrand is negligible in all but a single subregion, then the resulting one-sample integration is all but useless. Information, even very crude, allowing importance sampling to put many points in the active subregion would be much better than blind stratified sampling.

Stratified sampling really comes into its own if you have some way of estimating the variances, so that you can put unequal numbers of points in different subregions, according to (7.8.14) or its generalizations, *and* if you can find a way of dividing a region into a practical number of subregions (notably *not* K^d with large dimension d), while yet significantly reducing the variance of the function in each subregion compared to its variance in the full volume. Doing this requires a lot of knowledge about f, though different knowledge from what is required for importance sampling.

In practice, importance sampling and stratified sampling are not incompatible. In many, if not most, cases of interest, the integrand f is small everywhere in V except for a small fractional volume of "active regions." In these regions the magnitude of $|f|$ and the standard deviation $\sigma = [\mathrm{Var}\,(f)]^{1/2}$ are comparable in size, so both techniques will give about the same concentration of points. In more sophisticated implementations, it is also possible to "nest" the two techniques, so that (e.g.) importance sampling on a crude grid is followed by stratification within each grid cell.

Adaptive Monte Carlo: VEGAS

The VEGAS algorithm, invented by Peter Lepage [1,2], is widely used for multidimensional integrals that occur in elementary particle physics. VEGAS is primarily based on importance sampling, but it also does some stratified sampling if the dimension d is small enough to avoid K^d explosion (specifically, if $(K/2)^d < N/2$, with N the number of sample

points). The basic technique for importance sampling in VEGAS is to construct, adaptively, a multidimensional weight function g that is *separable*,

$$p \propto g(x, y, z, \ldots) = g_x(x) g_y(y) g_z(z) \ldots \tag{7.8.16}$$

Such a function avoids the K^d explosion in two ways: (i) It can be stored in the computer as d separate one-dimensional functions, each defined by K tabulated values, say — so that $K \times d$ replaces K^d. (ii) It can be sampled as a probability density by consecutively sampling the d one-dimensional functions to obtain coordinate vector components $(x, y, z, \ldots)$.

The optimal separable weight function can be shown to be [1]

$$g_x(x) \propto \left[\int dy \int dz \ldots \frac{f^2(x, y, z, \ldots)}{g_y(y) g_z(z) \ldots} \right]^{1/2} \tag{7.8.17}$$

(and correspondingly for $y, z, \ldots$). Notice that this reduces to $g \propto |f|$ (7.8.6) in one dimension. Equation (7.8.17) immediately suggests VEGAS' adaptive strategy: Given a set of g-functions (initially all constant, say), one samples the function f, accumulating not only the overall estimator of the integral, but also the Kd estimators (K subdivisions of the independent variable in each of d dimensions) of the right-hand side of equation (7.8.17). These then determine improved g functions for the next iteration.

When the integrand f is concentrated in one, or at most a few, regions in d-space, then the weight function g's quickly become large at coordinate values that are the projections of these regions onto the coordinate axes. The accuracy of the Monte Carlo integration is then enormously enhanced over what simple Monte Carlo would give.

The weakness of VEGAS is the obvious one: To the extent that the projection of the function f onto individual coordinate directions is uniform, VEGAS gives no concentration of sample points in those dimensions. The worst case for VEGAS, e.g., is an integrand that is concentrated close to a body diagonal line, e.g., one from $(0, 0, 0, \ldots)$ to $(1, 1, 1, \ldots)$. Since this geometry is completely nonseparable, VEGAS can give no advantage at all. More generally, VEGAS may not do well when the integrand is concentrated in one-dimensional (or higher) curved trajectories (or hypersurfaces), unless these happen to be oriented close to the coordinate directions.

The routine `vegas` that follows is essentially Lepage's standard version, minimally modified to conform to our conventions. (We thank Lepage for permission to reproduce the program here.) For consistency with other versions of the VEGAS algorithm in circulation, we have preserved original variable names. The parameter `NDMX` is what we have called K, the maximum number of increments along each axis; `MXDIM` is the maximum value of d; some other parameters are explained in the comments.

The `vegas` routine performs $m =$ `itmx` statistically independent evaluations of the desired integral, each with $N =$ `ncall` function evaluations. While statistically independent, these iterations do assist each other, since each one is used to refine the sampling grid for the next one. The results of all iterations are combined into a single best answer, and its estimated error, by the relations

$$I_{\rm best} = \sum_{i=1}^{m} \frac{I_i}{\sigma_i^2} \Bigg/ \sum_{i=1}^{m} \frac{1}{\sigma_i^2} \qquad \sigma_{\rm best} = \left(\sum_{i=1}^{m} \frac{1}{\sigma_i^2} \right)^{-1/2} \tag{7.8.18}$$

Also returned is the quantity

$$\chi^2/m \equiv \frac{1}{m-1} \sum_{i=1}^{m} \frac{(I_i - I_{\rm best})^2}{\sigma_i^2} \tag{7.8.19}$$

If this is significantly larger than 1, then the results of the iterations are statistically inconsistent, and the answers are suspect.

The input flag `init` can be used to advantage. One might have a call with `init=0`, `ncall=1000`, `itmx=5` immediately followed by a call with `init=1`, `ncall=100000`, `itmx=1`. The effect would be to develop a sampling grid over 5 iterations of a small number of samples, then to do a single high accuracy integration on the optimized grid.

Note that the user-supplied integrand function, `fxn`, has an argument `wgt` in addition to the expected evaluation point `x`. In most applications you ignore `wgt` inside the function. Occasionally, however, you may want to integrate some additional function or functions along with the principal function f. The integral of any such function g can be estimated by

$$I_g = \sum_i w_i g(\mathbf{x}) \tag{7.8.20}$$

where the w_i's and $\mathbf{x}$'s are the arguments `wgt` and `x`, respectively. It is straightforward to accumulate this sum inside your function `fxn`, and to pass the answer back to your main program via a common block. Of course, $g(\mathbf{x})$ had better resemble the principal function f to some degree, since the sampling will be optimized for f.

```
      SUBROUTINE vegas(region,ndim,fxn,init,ncall,itmx,nprn,
     *     tgral,sd,chi2a)
      INTEGER init,itmx,ncall,ndim,nprn,NDMX,MXDIM
      REAL tgral,chi2a,sd,region(2*ndim),fxn,ALPH,TINY
      PARAMETER (ALPH=1.5,NDMX=50,MXDIM=10,TINY=1.e-30)
      EXTERNAL fxn
C     USES fxn,ran2,rebin
```

Performs Monte Carlo integration of a user-supplied `ndim`-dimensional function `fxn` over a rectangular volume specified by `region`, a $2\times$`ndim` vector consisting of `ndim` "lower left" coordinates of the region followed by `ndim` "upper right" coordinates. The integration consists of `itmx` iterations, each with approximately `ncall` calls to the function. After each iteration the grid is refined; more than 5 or 10 iterations are rarely useful. The input flag `init` signals whether this call is a new start, or a subsequent call for additional iterations (see comments below). The input flag `nprn` (normally 0) controls the amount of diagnostic output. Returned answers are `tgral` (the best estimate of the integral), `sd` (its standard deviation), and `chi2a` (χ^2 per degree of freedom, an indicator of whether consistent results are being obtained). See text for further details.

```
      INTEGER i,idum,it,j,k,mds,nd,ndo,ng,npg,ia(MXDIM),kg(MXDIM)
      REAL calls,dv2g,dxg,f,f2,f2b,fb,rc,ti,tsi,wgt,xjac,xn,xnd,xo,
     *     d(NDMX,MXDIM),di(NDMX,MXDIM),dt(MXDIM),dx(MXDIM),
     *     r(NDMX),x(MXDIM),xi(NDMX,MXDIM),xin(NDMX),ran2
      DOUBLE PRECISION schi,si,swgt
      COMMON /ranno/ idum                 Means for random number initialization.
      SAVE                                Best make everything static, allowing restarts.
      if(init.le.0)then                   Normal entry. Enter here on a cold start.
        mds=1                               Change to mds=0 to disable stratified sampling, i.e., use im-
        ndo=1                                 portance sampling only.
        do 11 j=1,ndim
          xi(1,j)=1.
        enddo 11
      endif
      if (init.le.1)then                  Enter here to inherit the grid from a previous call, but not its
        si=0.d0                             answers.
        swgt=0.d0
        schi=0.d0
      endif
      if (init.le.2)then                  Enter here to inherit the previous grid and its answers.
        nd=NDMX
        ng=1
        if(mds.ne.0)then                  Set up for stratification.
          ng=(ncall/2.+0.25)**(1./ndim)
          mds=1
          if((2*ng-NDMX).ge.0)then
            mds=-1
            npg=ng/NDMX+1
            nd=ng/npg
            ng=npg*nd
          endif
        endif
        k=ng**ndim
```

```
      npg=max(ncall/k,2)
      calls=float(npg)*float(k)
      dxg=1./ng
      dv2g=(calls*dxg**ndim)**2/npg/npg/(npg-1.)
      xnd=nd
      dxg=dxg*xnd
      xjac=1./calls
      do 12 j=1,ndim
        dx(j)=region(j+ndim)-region(j)
        xjac=xjac*dx(j)
      enddo 12
      if(nd.ne.ndo)then        Do binning if necessary.
        do 13 i=1,max(nd,ndo)
          r(i)=1.
        enddo 13
        do 14 j=1,ndim
          call rebin(ndo/xnd,nd,r,xin,xi(1,j))
        enddo 14
        ndo=nd
      endif
      if(nprn.ge.0) write(*,200) ndim,calls,it,itmx,nprn,
*       ALPH,mds,nd,(j,region(j),j,region(j+ndim),j=1,ndim)
    endif
    do 28 it=1,itmx
      Main iteration loop. Can enter here (init ≥ 3) to do an additional itmx iterations with all
      other parameters unchanged.
      ti=0.
      tsi=0.
      do 16 j=1,ndim
        kg(j)=1
        do 15 i=1,nd
          d(i,j)=0.
          di(i,j)=0.
        enddo 15
      enddo 16
10    continue
        fb=0.
        f2b=0.
        do 19 k=1,npg
          wgt=xjac
          do 17 j=1,ndim
            xn=(kg(j)-ran2(idum))*dxg+1.
            ia(j)=max(min(int(xn),NDMX),1)
            if(ia(j).gt.1)then
              xo=xi(ia(j),j)-xi(ia(j)-1,j)
              rc=xi(ia(j)-1,j)+(xn-ia(j))*xo
            else
              xo=xi(ia(j),j)
              rc=(xn-ia(j))*xo
            endif
            x(j)=region(j)+rc*dx(j)
            wgt=wgt*xo*xnd
          enddo 17
          f=wgt*fxn(x,wgt)
          f2=f*f
          fb=fb+f
          f2b=f2b+f2
          do 18 j=1,ndim
            di(ia(j),j)=di(ia(j),j)+f
            if(mds.ge.0) d(ia(j),j)=d(ia(j),j)+f2
          enddo 18
        enddo 19
        f2b=sqrt(f2b*npg)
        f2b=(f2b-fb)*(f2b+fb)
```

```
            if (f2b.le.0.) f2b=TINY
            ti=ti+fb
            tsi=tsi+f2b
            if(mds.lt.0)then      Use stratified sampling.
                do 21 j=1,ndim
                    d(ia(j),j)=d(ia(j),j)+f2b
                enddo 21
            endif
        do 22 k=ndim,1,-1
            kg(k)=mod(kg(k),ng)+1
            if(kg(k).ne.1) goto 10
        enddo 22
        tsi=tsi*dv2g              Compute final results for this iteration.
        wgt=1./tsi
        si=si+dble(wgt)*dble(ti)
        schi=schi+dble(wgt)*dble(ti)**2
        swgt=swgt+dble(wgt)
        tgral=si/swgt
        chi2a=max((schi-si*tgral)/(it-.99d0),0.d0)
        sd=sqrt(1./swgt)
        tsi=sqrt(tsi)
        if(nprn.ge.0)then
            write(*,201) it,ti,tsi,tgral,sd,chi2a
            if(nprn.ne.0)then
                do 23 j=1,ndim
                    write(*,202) j,(xi(i,j),di(i,j),
*                        i=1+nprn/2,nd,nprn)
                enddo 23
            endif
        endif
        do 25 j=1,ndim            Refine the grid. Consult references to understand the subtlety
            xo=d(1,j)                 of this procedure. The refinement is damped, to avoid
            xn=d(2,j)                 rapid, destabilizing changes, and also compressed in range
            d(1,j)=(xo+xn)/2.         by the exponent ALPH.
            dt(j)=d(1,j)
            do 24 i=2,nd-1
                rc=xo+xn
                xo=xn
                xn=d(i+1,j)
                d(i,j)=(rc+xn)/3.
                dt(j)=dt(j)+d(i,j)
            enddo 24
            d(nd,j)=(xo+xn)/2.
            dt(j)=dt(j)+d(nd,j)
        enddo 25
        do 27 j=1,ndim
            rc=0.
            do 26 i=1,nd
                if(d(i,j).lt.TINY) d(i,j)=TINY
                r(i)=((1.-d(i,j)/dt(j))/(log(dt(j))-log(d(i,j))))**ALPH
                rc=rc+r(i)
            enddo 26
            call rebin(rc/xnd,nd,r,xin,xi(1,j))
        enddo 27
    enddo 28
    return
200 FORMAT(/' input parameters for vegas: ndim=',i3,' ncall=',f8.0
*       /28x,' it=',i5,' itmx=',i5
*       /28x,' nprn=',i3,' alph=',f5.2/28x,' mds=',i3,' nd=',i4
*       /(30x,'xl(',i2,')= ',g11.4,' xu(',i2,')= ',g11.4))
201 FORMAT(/' iteration no.',I3,': ','integral =',g14.7,'+/- ',g9.2
*       /' all iterations: integral =',g14.7,'+/- ',g9.2,
*       ' chi**2/it''n =',g9.2)
202 FORMAT(/' data for axis ',I2/' X delta i ',
```

```
*       ' x delta i ',' x delta i ',
*       /(1x,f7.5,1x,g11.4,5x,f7.5,1x,g11.4,5x,f7.5,1x,g11.4))
      END



      SUBROUTINE rebin(rc,nd,r,xin,xi)
      INTEGER nd
      REAL rc,r(*),xi(*),xin(*)
         Utility routine used by vegas, to rebin a vector of densities xi into new bins defined by
         a vector r.
      INTEGER i,k
      REAL dr,xn,xo
      k=0
      xo=0.
      dr=0.
      do 11 i=1,nd-1
1        if(rc.gt.dr)then
             k=k+1
             dr=dr+r(k)
         goto 1
         endif
         if(k.gt.1) xo=xi(k-1)
         xn=xi(k)
         dr=dr-rc
         xin(i)=xn-(xn-xo)*dr/r(k)
      enddo 11
      do 12 i=1,nd-1
         xi(i)=xin(i)
      enddo 12
      xi(nd)=1.
      return
      END
```

Recursive Stratified Sampling

The problem with stratified sampling, we have seen, is that it may not avoid the K^d explosion inherent in the obvious, Cartesian, tesselation of a d-dimensional volume. A technique called *recursive stratified sampling* [3] attempts to do this by successive bisections of a volume, not along all d dimensions, but rather along only one dimension at a time. The starting points are equations (7.8.10) and (7.8.13), applied to bisections of successively smaller subregions.

Suppose that we have a quota of N evaluations of the function f, and want to evaluate $\langle f \rangle'$ in the rectangular parallelepiped region $R = (\mathbf{x}_a, \mathbf{x}_b)$. (We denote such a region by the two coordinate vectors of its diagonally opposite corners.) First, we allocate a fraction p of N towards exploring the variance of f in R: We sample pN function values uniformly in R and accumulate the sums that will give the d different pairs of variances corresponding to the d different coordinate directions along which R can be bisected. In other words, in pN samples, we estimate $\mathrm{Var}\,(f)$ in each of the regions resulting from a possible bisection of R,

$$\begin{aligned} R_{ai} &\equiv (\mathbf{x}_a, \mathbf{x}_b - \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i) \\ R_{bi} &\equiv (\mathbf{x}_a + \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i, \mathbf{x}_b) \end{aligned} \tag{7.8.21}$$

Here $\mathbf{e}_i$ is the unit vector in the ith coordinate direction, $i = 1, 2, \ldots, d$.

Second, we inspect the variances to find the most favorable dimension i to bisect. By equation (7.8.15), we could, for example, choose that i for which the sum of the square roots of the variance estimators in regions R_{ai} and R_{bi} is minimized. (Actually, as we will explain, we do something slightly different.)

Third, we allocate the remaining $(1-p)N$ function evaluations between the regions R_{ai} and R_{bi}. If we used equation (7.8.15) to choose i, we should do this allocation according to equation (7.8.14).

We now have two parallelepipeds each with its own allocation of function evaluations for estimating the mean of f. Our "RSS" algorithm now shows itself to be *recursive*: To evaluate the mean in each region, we go back to the sentence beginning "First,..." in the paragraph above equation (7.8.21). (Of course, when the allocation of points to a region falls below some number, we resort to simple Monte Carlo rather than continue with the recursion.)

Finally, we combine the means, and also estimated variances of the two subvolumes, using equation (7.8.10) and the first line of equation (7.8.11).

This completes the RSS algorithm in its simplest form. Before we describe some additional tricks under the general rubric of "implementation details," we need to return briefly to equations (7.8.13)–(7.8.15) and derive the equations that we actually use instead of these. The right-hand side of equation (7.8.13) applies the familiar scaling law of equation (7.8.9) twice, once to a and again to b. This would be correct if the estimates $\langle f\rangle_a$ and $\langle f\rangle_b$ were each made by simple Monte Carlo, with uniformly random sample points. However, the two estimates of the mean are in fact made recursively. Thus, there is no reason to expect equation (7.8.9) to hold. Rather, we might substitute for equation (7.8.13) the relation,

$$\mathrm{Var}\left(\langle f\rangle'\right) = \frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N_a^\alpha} + \frac{\mathrm{Var}_b(f)}{(N-N_a)^\alpha}\right] \tag{7.8.22}$$

where α is an unknown constant ≥ 1 (the case of equality corresponding to simple Monte Carlo). In that case, a short calculation shows that $\mathrm{Var}\left(\langle f\rangle'\right)$ is minimized when

$$\frac{N_a}{N} = \frac{\mathrm{Var}_a(f)^{1/(1+\alpha)}}{\mathrm{Var}_a(f)^{1/(1+\alpha)} + \mathrm{Var}_b(f)^{1/(1+\alpha)}} \tag{7.8.23}$$

and that its minimum value is

$$\mathrm{Var}\left(\langle f\rangle'\right) \propto \left[\mathrm{Var}_a(f)^{1/(1+\alpha)} + \mathrm{Var}_b(f)^{1/(1+\alpha)}\right]^{1+\alpha} \tag{7.8.24}$$

Equations (7.8.22)–(7.8.24) reduce to equations (7.8.13)–(7.8.15) when $\alpha = 1$. Numerical experiments to find a self-consistent value for α find that $\alpha \approx 2$. That is, when equation (7.8.23) with $\alpha = 2$ is used recursively to allocate sample opportunities, the observed variance of the RSS algorithm goes approximately as N^{-2}, while any other value of α in equation (7.8.23) gives a poorer fall-off. (The sensitivity to α is, however, not very great; it is not known whether $\alpha = 2$ is an analytically justifiable result, or only a useful heuristic.)

Turn now to the routine, `miser`, which implements the RSS method. A bit of `FORTRAN` wizardry is its implementation of the required recursion. This is done by dimensioning an array `stack`, and a shorter "stack frame" `stf`; the latter has components that are equivalenced to variables that need to be preserved during the recursion, including a flag indicating where program control should return. A recursive call then consists of copying the stack frame onto the stack, incrementing the stack pointer `jstack`, and transferring control. A recursive return analogously pops the stack and transfers control to the saved location. Stack growth in `miser` is only logarithmic in N, since at each bifurcation one of the subvolumes can be processed immediately.

The principal difference between `miser`'s implementation and the algorithm as described thus far lies in how the variances on the right-hand side of equation (7.8.23) are estimated. We find empirically that it is somewhat more robust to use the square of the difference of maximum and minimum sampled function values, instead of the genuine second moment of the samples. This estimator is of course increasingly biased with increasing sample size; however, equation (7.8.23) uses it only to compare two subvolumes (a and b) having approximately equal numbers of samples. The "max minus min" estimator proves its worth when the preliminary sampling yields only a single point, or small number of points, in active regions of the integrand. In many realistic cases, these are indicators of nearby regions of even greater importance, and it is useful to let them attract the greater sampling weight that "max minus min" provides.

A second modification embodied in the code is the introduction of a "dithering parameter," `dith`, whose nonzero value causes subvolumes to be divided not exactly down the middle, but rather into fractions $0.5 \pm$ `dith`, with the sign of the $\pm$ randomly chosen by a built-in random number routine. Normally `dith` can be set to zero. However, there is a large advantage in taking `dith` to be nonzero if some special symmetry of the integrand puts the active region exactly at the midpoint of the region, or at the center of some power-of-two submultiple of the region. One wants to avoid the extreme case of the active region being evenly divided into 2^d abutting corners of a d-dimensional space. A typical nonzero value of `dith`, on those occasions when it is useful, might be 0.1. Of course, when the dithering parameter is nonzero, we must take the differing sizes of the subvolumes into account; the code does this through the variable `fracl`.

One final feature in the code deserves mention. The RSS algorithm uses a single set of sample points to evaluate equation (7.8.23) in all d directions. At bottom levels of the recursion, the number of sample points can be quite small. Although rare, it can happen that in one direction all the samples are in one half of the volume; in that case, that direction is ignored as a candidate for bifurcation. Even more rare is the possibility that all of the samples are in one half of the volume in *all* directions. In this case, a random direction is chosen. If this happens too often in your application, then you should increase `MNPT` (see line `if (jb.eq.0)...` in the code).

Note that `miser`, as given, returns as `ave` an estimate of the average function value $\langle\langle f \rangle\rangle$, not the integral of f over the region. The routine `vegas`, adopting the other convention, returns as `tgral` the integral. The two conventions are of course trivially related, by equation (7.8.8), since the volume V of the rectangular region is known.

```
      SUBROUTINE miser(func,region,ndim,npts,dith,ave,var)
      INTEGER ndim,npts,MNPT,MNBS,MAXD,NSTACK
      REAL ave,dith,var,region(2*ndim),func,TINY,BIG,PFAC
      PARAMETER (MNPT=15,MNBS=4*MNPT,MAXD=10,TINY=1.e-30,BIG=1.e30,
     *     NSTACK=1000,PFAC=0.1)
      EXTERNAL func
C     USES func,ranpt
```

Monte Carlo samples a user-supplied `ndim`-dimensional function `func` in a rectangular volume specified by `region`, a $2\times$`ndim` vector consisting of `ndim` "lower-left" coordinates of the region followed by `ndim` "upper-right" coordinates. The function is sampled a total of `npts` times, at locations determined by the method of recursive stratified sampling. The mean value of the function in the region is returned as `ave`; an estimate of the statistical uncertainty of `ave` (square of standard deviation) is returned as `var`. The input parameter `dith` should normally be set to zero, but can be set to (e.g.) 0.1 if `func`'s active region falls on the boundary of a power-of-two subdivision of `region`.

Parameters: `PFAC` is the fraction of remaining function evaluations used *at each stage* to explore the variance of `func`. At least `MNPT` function evaluations are performed in any terminal subregion; a subregion is further bisected only if at least `MNBS` function evaluations are available. `MAXD` is the largest value of `ndim`. `NSTACK` is the total size of the stack.

```
      INTEGER iran,j,jb,jstack,n,naddr,np,npre,nptl,nptr,nptt
      REAL avel,fracl,fval,rgl,rgm,rgr,s,sigl,siglb,sigr,sigrb,sum,
     *     sumb,summ,summ2,varl,fmaxl(MAXD),fmaxr(MAXD),fminl(MAXD),
     *     fminr(MAXD),pt(MAXD),rmid(MAXD),stack(NSTACK),stf(9)
      EQUIVALENCE (stf(1),avel),(stf(2),varl),(stf(3),jb),
     *     (stf(4),nptr),(stf(5),naddr),(stf(6),rgl),(stf(7),rgm),
     *     (stf(8),rgr),(stf(9),fracl)
      SAVE iran
      DATA iran /0/
      jstack=0
      nptt=npts
1     continue
      if (nptt.lt.MNBS) then        Too few points to bisect; do straight Monte Carlo.
        np=abs(nptt)
        summ=0.
        summ2=0.
        do 11 n=1,np
          call ranpt(pt,region,ndim)
```

```
            fval=func(pt)
            summ=summ+fval
            summ2=summ2+fval**2
        enddo 11
        ave=summ/np
        var=max(TINY,(summ2-summ**2/np)/np**2)
    else                        Do the preliminary (uniform) sampling.
        npre=max(int(nptt*PFAC),MNPT)
        do 12 j=1,ndim          Initialize the left and right bounds for each dimension.
            iran=mod(iran*2661+36979,175000)
            s=sign(dith,float(iran-87500))
            rmid(j)=(0.5+s)*region(j)+(0.5-s)*region(j+ndim)
            fminl(j)=BIG
            fminr(j)=BIG
            fmaxl(j)=-BIG
            fmaxr(j)=-BIG
        enddo 12
        do 14 n=1,npre          Loop over the points in the sample.
            call ranpt(pt,region,ndim)
            fval=func(pt)
            do 13 j=1,ndim      Find the left and right bounds for each dimension.
                if(pt(j).le.rmid(j))then
                    fminl(j)=min(fminl(j),fval)
                    fmaxl(j)=max(fmaxl(j),fval)
                else
                    fminr(j)=min(fminr(j),fval)
                    fmaxr(j)=max(fmaxr(j),fval)
                endif
            enddo 13
        enddo 14
        sumb=BIG                Choose which dimension jb to bisect.
        jb=0
        siglb=1.
        sigrb=1.
        do 15 j=1,ndim
            if(fmaxl(j).gt.fminl(j).and.fmaxr(j).gt.fminr(j))then
                sigl=max(TINY,(fmaxl(j)-fminl(j))**(2./3.))
                sigr=max(TINY,(fmaxr(j)-fminr(j))**(2./3.))
                sum=sigl+sigr   Equation (7.8.24), see text.
                if (sum.le.sumb) then
                    sumb=sum
                    jb=j
                    siglb=sigl
                    sigrb=sigr
                endif
            endif
        enddo 15
        if (jb.eq.0) jb=1+(ndim*iran)/175000      MNPT may be too small.
        rgl=region(jb)          Apportion the remaining points between left and right.
        rgm=rmid(jb)
        rgr=region(jb+ndim)
        fracl=abs((rgm-rgl)/(rgr-rgl))
        nptl=MNPT+(nptt-npre-2*MNPT)
*            *fracl*siglb/(fracl*siglb+(1.-fracl)*sigrb)        Equation (7.8.23).
        nptr=nptt-npre-nptl
        region(jb+ndim)=rgm     Set region to left.
        naddr=1                 Push the stack.
        do 16 j=1,9
            stack(jstack+j)=stf(j)
        enddo 16
        jstack=jstack+9
        nptt=nptl
        goto 1                  Dispatch recursive call; will return back here eventually.
10      continue
```

```
        avel=ave                    Save left estimates on stack variable.
        varl=var
        region(jb)=rgm              Set region to right.
        region(jb+ndim)=rgr
        naddr=2                     Push the stack.
        do 17 j=1,9
            stack(jstack+j)=stf(j)
        enddo 17
        jstack=jstack+9
        nptt=nptr
        goto 1                      Dispatch recursive call; will return back here eventually.
20      continue
        region(jb)=rgl                  Restore region to original value (so that we don't
        ave=fracl*avel+(1.-fracl)*ave       need to include it on the stack).
        var=fracl**2*varl+(1.-fracl)**2*var     Combine left and right regions by equa-
    endif                                           tion (7.8.11) (1st line).
    if (jstack.ne.0) then           Pop the stack.
        jstack=jstack-9
        do 18 j=1,9
            stf(j)=stack(jstack+j)
        enddo 18
        goto (10,20),naddr
        pause 'miser: never get here'
    endif
    return
    END
```

The `miser` routine calls a short subroutine `ranpt` to get a random point within a specified d-dimensional region. The following version of `ranpt` makes consecutive calls to a uniform random number generator and does the obvious scaling. One can easily modify `ranpt` to generate its points via the quasi-random routine `sobseq` (§7.7). We find that `miser` with `sobseq` can be considerably more accurate than `miser` with uniform random deviates. Since the use of RSS and the use of quasi-random numbers are completely separable, however, we have not made the code given here dependent on `sobseq`. A similar remark might be made regarding importance sampling, which could in principle be combined with RSS. (One could in principle combine `vegas` and `miser`, although the programming would be intricate.)

```
    SUBROUTINE ranpt(pt,region,n)
    INTEGER n,idum
    REAL pt(n),region(2*n)
    COMMON /ranno/ idum
    SAVE /ranno/
C   USES ran1
        Returns a uniformly random point pt in an n-dimensional rectangular region. Used by
        miser; calls ran1 for uniform deviates. Your main program should initialize idum, through
        the COMMON block /ranno/, to a negative seed integer.
    INTEGER j
    REAL ran1
    do 11 j=1,n
        pt(j)=region(j)+(region(j+n)-region(j))*ran1(idum)
    enddo 11
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Hammersley, J.M. and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).

Kalos, M.H. and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag).

Lepage, G.P. 1978, *Journal of Computational Physics*, vol. 27, pp. 192–203. [1]

Lepage, G.P. 1980, "VEGAS: An Adaptive Multidimensional Integration Program," Publication CLNS-80/447, Cornell University. [2]

Press, W.H., and Farrar, G.R. 1990, *Computers in Physics*, vol. 4, pp. 190–195. [3]

Chapter 8. Sorting

8.0 Introduction

This chapter almost doesn't belong in a book on *numerical* methods. However, some practical knowledge of techniques for sorting is an indispensable part of any good programmer's expertise. We would not want you to consider yourself expert in numerical techniques while remaining ignorant of so basic a subject.

In conjunction with numerical work, sorting is frequently necessary when data (either experimental or numerically generated) are being handled. One has tables or lists of numbers, representing one or more independent (or "control") variables, and one or more dependent (or "measured") variables. One may wish to arrange these data, in various circumstances, in order by one or another of these variables. Alternatively, one may simply wish to identify the "median" value, or the "upper quartile" value of one of the lists of values. This task, closely related to sorting, is called *selection*.

Here, more specifically, are the tasks that this chapter will deal with:

- Sort, i.e., rearrange, an array of numbers into numerical order.
- Rearrange an array into numerical order while performing the corresponding rearrangement of one or more additional arrays, so that the correspondence between elements in all arrays is maintained.
- Given an array, prepare an *index table* for it, i.e., a table of pointers telling which number array element comes first in numerical order, which second, and so on.
- Given an array, prepare a *rank table* for it, i.e., a table telling what is the numerical rank of the first array element, the second array element, and so on.
- Select the Mth largest element from an array.

For the basic task of sorting N elements, the best algorithms require on the order of several times $N \log_2 N$ operations. The algorithm inventor tries to reduce the constant in front of this estimate to as small a value as possible. Two of the best algorithms are *Quicksort* (§8.2), invented by the inimitable C.A.R. Hoare, and *Heapsort* (§8.3), invented by J.W.J. Williams.

For large N (say > 1000), Quicksort is faster, on most machines, by a factor of 1.5 or 2; it requires a bit of extra memory, however, and is a moderately complicated program. Heapsort is a true "sort in place," and is somewhat more compact to program and therefore a bit easier to modify for special purposes. On balance, we recommend Quicksort because of its speed, but we implement both routines.

For small N one does better to use an algorithm whose operation count goes as a higher, i.e., poorer, power of N, if the constant in front is small enough. For $N < 20$, roughly, the method of *straight insertion* (§8.1) is concise and fast enough. We include it with some trepidation: It is an N^2 algorithm, whose potential for misuse (by using it for too large an N) is great. The resultant waste of computer time is so awesome, that we were tempted not to include any N^2 routine at all. We *will* draw the line, however, at the inefficient N^2 algorithm, beloved of elementary computer science texts, called *bubble sort*. If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!

For $N < 50$, roughly, *Shell's method* (§8.1), only slightly more complicated to program than straight insertion, is competitive with the more complicated Quicksort on many machines. This method goes as $N^{3/2}$ in the worst case, but is usually faster.

See references [1,2] for further information on the subject of sorting, and for detailed references to the literature.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley). [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapters 8–13. [2]

8.1 Straight Insertion and Shell's Method

Straight insertion is an N^2 routine, and should be used only for small N, say < 20.

The technique is exactly the one used by experienced card players to sort their cards: Pick out the second card and put it in order with respect to the first; then pick out the third card and insert it into the sequence among the first two; and so on until the last card has been picked out and inserted.

```
      SUBROUTINE piksrt(n,arr)
      INTEGER n
      REAL arr(n)
        Sorts an array arr(1:n) into ascending numerical order, by straight insertion. n is input;
        arr is replaced on output by its sorted rearrangement.
      INTEGER i,j
      REAL a
      do 12 j=2,n                    Pick out each element in turn.
        a=arr(j)
        do 11 i=j-1,1,-1             Look for the place to insert it.
          if(arr(i).le.a)goto 10
          arr(i+1)=arr(i)
        enddo 11
        i=0
10      arr(i+1)=a                   Insert it.
      enddo 12
      return
      END
```

What if you also want to rearrange an array `brr` at the same time as you sort `arr`? Simply move an element of `brr` whenever you move an element of `arr`:

For small N one does better to use an algorithm whose operation count goes as a higher, i.e., poorer, power of N, if the constant in front is small enough. For $N < 20$, roughly, the method of *straight insertion* (§8.1) is concise and fast enough. We include it with some trepidation: It is an N^2 algorithm, whose potential for misuse (by using it for too large an N) is great. The resultant waste of computer time is so awesome, that we were tempted not to include any N^2 routine at all. We *will* draw the line, however, at the inefficient N^2 algorithm, beloved of elementary computer science texts, called *bubble sort*. If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!

For $N < 50$, roughly, *Shell's method* (§8.1), only slightly more complicated to program than straight insertion, is competitive with the more complicated Quicksort on many machines. This method goes as $N^{3/2}$ in the worst case, but is usually faster.

See references [1,2] for further information on the subject of sorting, and for detailed references to the literature.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley). [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapters 8–13. [2]

8.1 Straight Insertion and Shell's Method

Straight insertion is an N^2 routine, and should be used only for small N, say < 20.

The technique is exactly the one used by experienced card players to sort their cards: Pick out the second card and put it in order with respect to the first; then pick out the third card and insert it into the sequence among the first two; and so on until the last card has been picked out and inserted.

```
      SUBROUTINE piksrt(n,arr)
      INTEGER n
      REAL arr(n)
        Sorts an array arr(1:n) into ascending numerical order, by straight insertion. n is input;
        arr is replaced on output by its sorted rearrangement.
      INTEGER i,j
      REAL a
      do 12 j=2,n                    Pick out each element in turn.
        a=arr(j)
        do 11 i=j-1,1,-1             Look for the place to insert it.
          if(arr(i).le.a)goto 10
          arr(i+1)=arr(i)
        enddo 11
        i=0
10      arr(i+1)=a                   Insert it.
      enddo 12
      return
      END
```

What if you also want to rearrange an array `brr` at the same time as you sort `arr`? Simply move an element of `brr` whenever you move an element of `arr`:

```
      SUBROUTINE piksr2(n,arr,brr)
      INTEGER n
      REAL arr(n),brr(n)
          Sorts an array arr(1:n) into ascending numerical order, by straight insertion, while making
          the corresponding rearrangement of the array brr(1:n).
      INTEGER i,j
      REAL a,b
      do 12 j=2,n                    Pick out each element in turn.
          a=arr(j)
          b=brr(j)
          do 11 i=j-1,1,-1           Look for the place to insert it.
              if(arr(i).le.a)goto 10
              arr(i+1)=arr(i)
              brr(i+1)=brr(i)
          enddo 11
          i=0
10        arr(i+1)=a                 Insert it.
          brr(i+1)=b
      enddo 12
      return
      END
```

For the case of rearranging a larger number of arrays by sorting on one of them, see §8.4.

Shell's Method

This is actually a variant on straight insertion, but a very powerful variant indeed. The rough idea, e.g., for the case of sorting 16 numbers $n_1 \ldots n_{16}$, is this: First sort, by straight insertion, each of the 8 groups of 2 (n_1, n_9), (n_2, n_{10}), $\ldots$, (n_8, n_{16}). Next, sort each of the 4 groups of 4 (n_1, n_5, n_9, n_{13}), $\ldots$, $(n_4, n_8, n_{12}, n_{16})$. Next sort the 2 groups of 8 records, beginning with $(n_1, n_3, n_5, n_7, n_9, n_{11}, n_{13}, n_{15})$. Finally, sort the whole list of 16 numbers.

Of course, only the *last* sort is *necessary* for putting the numbers into order. So what is the purpose of the previous partial sorts? The answer is that the previous sorts allow numbers efficiently to filter up or down to positions close to their final resting places. Therefore, the straight insertion passes on the final sort rarely have to go past more than a "few" elements before finding the right place. (Think of sorting a hand of cards that are already almost in order.)

The spacings between the numbers sorted on each pass through the data (8,4,2,1 in the above example) are called the *increments*, and a Shell sort is sometimes called a *diminishing increment sort*. There has been a lot of research into how to choose a good set of increments, but the optimum choice is not known. The set $\ldots, 8, 4, 2, 1$ is in fact not a good choice, especially for N a power of 2. A much better choice is the sequence

$$(3^k - 1)/2, \ldots, 40, 13, 4, 1 \tag{8.1.1}$$

which can be generated by the recurrence

$$i_1 = 1, \qquad i_{k+1} = 3i_k + 1, \quad k = 1, 2, \ldots \tag{8.1.2}$$

It can be shown (see [1]) that for this sequence of increments the number of operations required in all is of order $N^{3/2}$ for the worst possible ordering of the original data.

For "randomly" ordered data, the operations count goes approximately as $N^{1.25}$, at least for $N < 60000$. For $N > 50$, however, Quicksort is generally faster. The program follows:

```
      SUBROUTINE shell(n,a)
      INTEGER n
      REAL a(n)
         Sorts an array a(1:n) into ascending numerical order by Shell's method (diminishing in-
         crement sort). n is input; a is replaced on output by its sorted rearrangement.
      INTEGER i,j,inc
      REAL v
      inc=1                                  Determine the starting increment.
1     inc=3*inc+1
      if(inc.le.n)goto 1
2     continue                               Loop over the partial sorts.
          inc=inc/3
          do 11 i=inc+1,n                    Outer loop of straight insertion.
              v=a(i)
              j=i
3             if(a(j-inc).gt.v)then          Inner loop of straight insertion.
                  a(j)=a(j-inc)
                  j=j-inc
                  if(j.le.inc)goto 4
              goto 3
              endif
4             a(j)=v
          enddo 11
      if(inc.gt.1)goto 2
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 8.

8.2 Quicksort

Quicksort is, on most machines, on average, for large N, the fastest known sorting algorithm. It is a "partition-exchange" sorting method: A "partitioning element" a is selected from the array. Then by pairwise exchanges of elements, the original array is partitioned into two subarrays. At the end of a round of partitioning, the element a is in its final place in the array. All elements in the left subarray are $\leq$ a, while all elements in the right subarray are $\geq$ a. The process is then repeated on the left and right subarrays independently, and so on.

The partitioning process is carried out by selecting some element, say the leftmost, as the partitioning element a. Scan a pointer up the array until you find an element $>$ a, and then scan another pointer down from the end of the array until you find an element $<$ a. These two elements are clearly out of place for the final partitioned array, so exchange them. Continue this process until the pointers

For "randomly" ordered data, the operations count goes approximately as $N^{1.25}$, at least for $N < 60000$. For $N > 50$, however, Quicksort is generally faster. The program follows:

```
      SUBROUTINE shell(n,a)
      INTEGER n
      REAL a(n)
         Sorts an array a(1:n) into ascending numerical order by Shell's method (diminishing in-
         crement sort). n is input; a is replaced on output by its sorted rearrangement.
      INTEGER i,j,inc
      REAL v
      inc=1                                  Determine the starting increment.
1     inc=3*inc+1
      if(inc.le.n)goto 1
2     continue                               Loop over the partial sorts.
          inc=inc/3
          do 11 i=inc+1,n                    Outer loop of straight insertion.
              v=a(i)
              j=i
3             if(a(j-inc).gt.v)then          Inner loop of straight insertion.
                  a(j)=a(j-inc)
                  j=j-inc
                  if(j.le.inc)goto 4
              goto 3
              endif
4             a(j)=v
          enddo 11
      if(inc.gt.1)goto 2
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 8.

8.2 Quicksort

Quicksort is, on most machines, on average, for large N, the fastest known sorting algorithm. It is a "partition-exchange" sorting method: A "partitioning element" a is selected from the array. Then by pairwise exchanges of elements, the original array is partitioned into two subarrays. At the end of a round of partitioning, the element a is in its final place in the array. All elements in the left subarray are $\leq$ a, while all elements in the right subarray are $\geq$ a. The process is then repeated on the left and right subarrays independently, and so on.

The partitioning process is carried out by selecting some element, say the leftmost, as the partitioning element a. Scan a pointer up the array until you find an element $>$ a, and then scan another pointer down from the end of the array until you find an element $<$ a. These two elements are clearly out of place for the final partitioned array, so exchange them. Continue this process until the pointers

cross. This is the right place to insert a, and that round of partitioning is done. The question of the best strategy when an element is equal to the partitioning element is subtle; we refer you to Sedgewick [1] for a discussion. (Answer: You should stop and do an exchange.)

Quicksort requires an auxiliary array of storage, of length $2\log_2 N$, which it uses as a push-down stack for keeping track of the pending subarrays. When a subarray has gotten down to some size M, it becomes faster to sort it by straight insertion (§8.1), so we will do this. The optimal setting of M is machine dependent, but $M = 7$ is not too far wrong. Some people advocate leaving the short subarrays unsorted until the end, and then doing one giant insertion sort at the end. Since each element moves at most 7 places, this is just as efficient as doing the sorts immediately, and saves on the overhead. However, on modern machines with paged memory, there is increased overhead when dealing with a large array all at once. We have not found any advantage in saving the insertion sorts till the end.

As already mentioned, Quicksort's *average* running time is fast, but its *worst case* running time can be very slow: For the worst case it is, in fact, an N^2 method! And for the most straightforward implementation of Quicksort it turns out that the worst case is achieved for an input array that is already in order! This ordering of the input array might easily occur in practice. One way to avoid this is to use a little random number generator to choose a random element as the partitioning element. Another is to use instead the median of the first, middle, and last elements of the current subarray.

The great speed of Quicksort comes from the simplicity and efficiency of its inner loop. Simply adding one unnecessary test (for example, a test that your pointer has not moved off the end of the array) can almost double the running time! One avoids such unnecessary tests by placing "sentinels" at either end of the subarray being partitioned. The leftmost sentinel is $\leq$ a, the rightmost $\geq$ a. With the "median-of-three" selection of a partitioning element, we can use the two elements that were not the median to be the sentinels for that subarray.

Our implementation closely follows [1]:

```
      SUBROUTINE sort(n,arr)
      INTEGER n,M,NSTACK
      REAL arr(n)
      PARAMETER (M=7,NSTACK=50)
          Sorts an array arr(1:n) into ascending numerical order using the Quicksort algorithm. n
          is input; arr is replaced on output by its sorted rearrangement.
          Parameters: M is the size of subarrays sorted by straight insertion and NSTACK is the required
          auxiliary storage.
      INTEGER i,ir,j,jstack,k,l,istack(NSTACK)
      REAL a,temp
      jstack=0
      l=1
      ir=n
1     if(ir-l.lt.M)then                    Insertion sort when subarray small enough.
          do 12 j=l+1,ir
              a=arr(j)
              do 11 i=j-1,l,-1
                  if(arr(i).le.a)goto 2
                  arr(i+1)=arr(i)
              enddo 11
              i=l-1
2             arr(i+1)=a
          enddo 12
```

```
        if(jstack.eq.0)return
        ir=istack(jstack)           Pop stack and begin a new round of partitioning.
        l=istack(jstack-1)
        jstack=jstack-2
    else
        k=(l+ir)/2                  Choose median of left, center, and right elements as par-
        temp=arr(k)                     titioning element a. Also rearrange so that a(l) ≤
        arr(k)=arr(l+1)                 a(l+1) ≤ a(ir).
        arr(l+1)=temp
        if(arr(l).gt.arr(ir))then
            temp=arr(l)
            arr(l)=arr(ir)
            arr(ir)=temp
        endif
        if(arr(l+1).gt.arr(ir))then
            temp=arr(l+1)
            arr(l+1)=arr(ir)
            arr(ir)=temp
        endif
        if(arr(l).gt.arr(l+1))then
            temp=arr(l)
            arr(l)=arr(l+1)
            arr(l+1)=temp
        endif
        i=l+1                       Initialize pointers for partitioning.
        j=ir
        a=arr(l+1)                  Partitioning element.
3       continue                    Beginning of innermost loop.
            i=i+1                   Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
            j=j-1                   Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5            Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                 Exchange elements.
        arr(i)=arr(j)
        arr(j)=temp
        goto 3                      End of innermost loop.
5       arr(l+1)=arr(j)             Insert partitioning element.
        arr(j)=a
        jstack=jstack+2
          Push pointers to larger subarray on stack, process smaller subarray immediately.
        if(jstack.gt.NSTACK)pause 'NSTACK too small in sort'
        if(ir-i+1.ge.j-l)then
            istack(jstack)=ir
            istack(jstack-1)=i
            ir=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        endif
    endif
    goto 1
    END
```

As usual you can move any other arrays around at the same time as you sort `arr`. At the risk of being repetitious:

```
      SUBROUTINE sort2(n,arr,brr)
      INTEGER n,M,NSTACK
      REAL arr(n),brr(n)
      PARAMETER (M=7,NSTACK=50)
        Sorts an array arr(1:n) into ascending order using Quicksort, while making the corresponding rearrangement of the array brr(1:n).
      INTEGER i,ir,j,jstack,k,l,istack(NSTACK)
      REAL a,b,temp
      jstack=0
      l=1
      ir=n
1     if(ir-l.lt.M)then                 Insertion sort when subarray small enough.
        do 12 j=l+1,ir
          a=arr(j)
          b=brr(j)
          do 11 i=j-1,l,-1
            if(arr(i).le.a)goto 2
            arr(i+1)=arr(i)
            brr(i+1)=brr(i)
          enddo 11
          i=l-1
2         arr(i+1)=a
          brr(i+1)=b
        enddo 12
        if(jstack.eq.0)return
        ir=istack(jstack)               Pop stack and begin a new round of partitioning.
        l=istack(jstack-1)
        jstack=jstack-2
      else
        k=(l+ir)/2                      Choose median of left, center and right elements as partitioning element a. Also rearrange so that a(l) ≤ a(l+1) ≤ a(ir).
        temp=arr(k)
        arr(k)=arr(l+1)
        arr(l+1)=temp
        temp=brr(k)
        brr(k)=brr(l+1)
        brr(l+1)=temp
        if(arr(l).gt.arr(ir))then
          temp=arr(l)
          arr(l)=arr(ir)
          arr(ir)=temp
          temp=brr(l)
          brr(l)=brr(ir)
          brr(ir)=temp
        endif
        if(arr(l+1).gt.arr(ir))then
          temp=arr(l+1)
          arr(l+1)=arr(ir)
          arr(ir)=temp
          temp=brr(l+1)
          brr(l+1)=brr(ir)
          brr(ir)=temp
        endif
        if(arr(l).gt.arr(l+1))then
          temp=arr(l)
          arr(l)=arr(l+1)
          arr(l+1)=temp
          temp=brr(l)
          brr(l)=brr(l+1)
          brr(l+1)=temp
        endif
        i=l+1                           Initialize pointers for partitioning.
        j=ir
        a=arr(l+1)                      Partitioning element.
        b=brr(l+1)
```

```
3       continue                        Beginning of innermost loop.
            i=i+1                       Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
            j=j-1                       Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5                Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                     Exchange elements of both arrays.
        arr(i)=arr(j)
        arr(j)=temp
        temp=brr(i)
        brr(i)=brr(j)
        brr(j)=temp
        goto 3                          End of innermost loop.
5       arr(l+1)=arr(j)                 Insert partitioning element in both arrays.
        arr(j)=a
        brr(l+1)=brr(j)
        brr(j)=b
        jstack=jstack+2
          Push pointers to larger subarray on stack, process smaller subarray immediately.
        if(jstack.gt.NSTACK)pause 'NSTACK too small in sort2'
        if(ir-i+1.ge.j-l)then
            istack(jstack)=ir
            istack(jstack-1)=i
            ir=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        endif
    endif
    goto 1
    END
```

You could, in principle, rearrange any number of additional arrays along with `brr`, but this becomes wasteful as the number of such arrays becomes large. The preferred technique is to make use of an index table, as described in §8.4.

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847–857. [1]

8.3 Heapsort

While usually not quite as fast as Quicksort, Heapsort is one of our favorite sorting routines. It is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ process, not only on average, but also for the worst-case order of input data. In fact, its worst case is only 20 percent or so worse than its average running time.

It is beyond our scope to give a complete exposition on the theory of Heapsort. We will mention the general principles, then let you refer to the references [1,2], or analyze the program yourself, if you want to understand the details.

A set of N numbers $a_i,\ i = 1, \ldots, N$, is said to form a "heap" if it satisfies the relation

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$

```
3       continue                        Beginning of innermost loop.
            i=i+1                       Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
            j=j-1                       Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5                Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                     Exchange elements of both arrays.
        arr(i)=arr(j)
        arr(j)=temp
        temp=brr(i)
        brr(i)=brr(j)
        brr(j)=temp
        goto 3                          End of innermost loop.
5       arr(l+1)=arr(j)                 Insert partitioning element in both arrays.
        arr(j)=a
        brr(l+1)=brr(j)
        brr(j)=b
        jstack=jstack+2
          Push pointers to larger subarray on stack, process smaller subarray immediately.
        if(jstack.gt.NSTACK)pause 'NSTACK too small in sort2'
        if(ir-i+1.ge.j-l)then
            istack(jstack)=ir
            istack(jstack-1)=i
            ir=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        endif
    endif
    goto 1
    END
```

You could, in principle, rearrange any number of additional arrays along with `brr`, but this becomes wasteful as the number of such arrays becomes large. The preferred technique is to make use of an index table, as described in §8.4.

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847–857. [1]

8.3 Heapsort

While usually not quite as fast as Quicksort, Heapsort is one of our favorite sorting routines. It is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ process, not only on average, but also for the worst-case order of input data. In fact, its worst case is only 20 percent or so worse than its average running time.

It is beyond our scope to give a complete exposition on the theory of Heapsort. We will mention the general principles, then let you refer to the references [1,2], or analyze the program yourself, if you want to understand the details.

A set of N numbers $a_i,\ i = 1, \ldots, N$, is said to form a "heap" if it satisfies the relation

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$

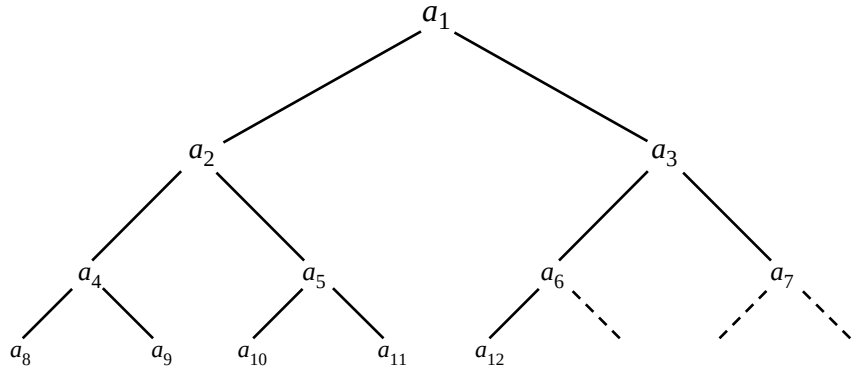


Figure 8.3.1. Ordering implied by a "heap," here of 12 elements. Elements connected by an upward path are sorted with respect to one another, but there is not necessarily any ordering among elements related only "laterally."

Here the division in $j/2$ means "integer divide," i.e., is an exact integer or else is rounded down to the closest integer. Definition (8.3.1) will make sense if you think of the numbers a_i as being arranged in a binary tree, with the top, "boss," node being a_1, the two "underling" nodes being a_2 and a_3, *their* four underling nodes being a_4 through a_7, etc. (See Figure 8.3.1.) In this form, a heap has every "supervisor" greater than or equal to its two "supervisees," down through the levels of the hierarchy.

If you have managed to rearrange your array into an order that forms a heap, then sorting it is very easy: You pull off the "top of the heap," which will be the largest element yet unsorted. Then you "promote" to the top of the heap its largest underling. Then you promote *its* largest underling, and so on. The process is like what happens (or is supposed to happen) in a large corporation when the chairman of the board retires. You then repeat the whole process by retiring the new chairman of the board. Evidently the whole thing is an $N \log_2 N$ process, since each retiring chairman leads to $\log_2 N$ promotions of underlings.

Well, how do you arrange the array into a heap in the first place? The answer is again a "sift-up" process like corporate promotion. Imagine that the corporation starts out with $N/2$ employees on the production line, but with no supervisors. Now a supervisor is hired to supervise two workers. If he is less capable than one of his workers, that one is promoted in his place, and he joins the production line. After supervisors are hired, then supervisors of supervisors are hired, and so on up the corporate ladder. Each employee is brought in at the top of the tree, but then immediately sifted down, with more capable workers promoted until their proper corporate level has been reached.

In the Heapsort implementation, the same "sift-up" code can be used for the initial creation of the heap and for the subsequent retirement-and-promotion phase. One execution of the Heapsort subroutine represents the entire life-cycle of a giant corporation: $N/2$ workers are hired; $N/2$ potential supervisors are hired; there is a sifting up in the ranks, a sort of super Peter Principle: in due course, each of the original employees gets promoted to chairman of the board.

```
      SUBROUTINE hpsort(n,ra)
      INTEGER n
      REAL ra(n)
        Sorts an array ra(1:n) into ascending numerical order using the Heapsort algorithm. n is
        input; ra is replaced on output by its sorted rearrangement.
      INTEGER i,ir,j,l
      REAL rra
      if (n.lt.2) return
        The index l will be decremented from its initial value down to 1 during the "hiring" (heap
        creation) phase. Once it reaches 1, the index ir will be decremented from its initial value
        down to 1 during the "retirement-and-promotion" (heap selection) phase.
      l=n/2+1
      ir=n
10    continue
        if(l.gt.1)then                Still in hiring phase.
          l=l-1
          rra=ra(l)
        else                          In retirement-and-promotion phase.
          rra=ra(ir)                  Clear a space at end of array.
          ra(ir)=ra(1)                Retire the top of the heap into it.
          ir=ir-1                     Decrease the size of the corporation.
          if(ir.eq.1)then             Done with the last promotion.
            ra(1)=rra                 The least competent worker of all!
            return
          endif
        endif
        i=l                           Whether in the hiring phase or promotion phase, we here
        j=l+l                           set up to sift down element rra to its proper level.
20      if(j.le.ir)then               "Do while j.le.ir:"
          if(j.lt.ir)then
            if(ra(j).lt.ra(j+1))j=j+1     Compare to the better underling.
          endif
          if(rra.lt.ra(j))then        Demote rra.
            ra(i)=ra(j)
            i=j
            j=j+j
          else                        This is rra's level. Set j to terminate the sift-down.
            j=ir+1
          endif
        goto 20
        endif
        ra(i)=rra                     Put rra into its slot.
      goto 10
      END
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 11. [2]

8.4 Indexing and Ranking

The concept of *keys* plays a prominent role in the management of data files. A data *record* in such a file may contain several items, or *fields*. For example, a record in a file of weather observations may have fields recording time, temperature, and

```
      SUBROUTINE hpsort(n,ra)
      INTEGER n
      REAL ra(n)
        Sorts an array ra(1:n) into ascending numerical order using the Heapsort algorithm. n is
        input; ra is replaced on output by its sorted rearrangement.
      INTEGER i,ir,j,l
      REAL rra
      if (n.lt.2) return
        The index l will be decremented from its initial value down to 1 during the "hiring" (heap
        creation) phase. Once it reaches 1, the index ir will be decremented from its initial value
        down to 1 during the "retirement-and-promotion" (heap selection) phase.
      l=n/2+1
      ir=n
10    continue
        if(l.gt.1)then                Still in hiring phase.
          l=l-1
          rra=ra(l)
        else                          In retirement-and-promotion phase.
          rra=ra(ir)                  Clear a space at end of array.
          ra(ir)=ra(1)                Retire the top of the heap into it.
          ir=ir-1                     Decrease the size of the corporation.
          if(ir.eq.1)then             Done with the last promotion.
            ra(1)=rra                 The least competent worker of all!
            return
          endif
        endif
        i=l                           Whether in the hiring phase or promotion phase, we here
        j=l+l                           set up to sift down element rra to its proper level.
20      if(j.le.ir)then               "Do while j.le.ir:"
          if(j.lt.ir)then
            if(ra(j).lt.ra(j+1))j=j+1     Compare to the better underling.
          endif
          if(rra.lt.ra(j))then        Demote rra.
            ra(i)=ra(j)
            i=j
            j=j+j
          else                        This is rra's level. Set j to terminate the sift-down.
            j=ir+1
          endif
        goto 20
        endif
        ra(i)=rra                     Put rra into its slot.
      goto 10
      END
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 11. [2]

8.4 Indexing and Ranking

The concept of *keys* plays a prominent role in the management of data files. A data *record* in such a file may contain several items, or *fields*. For example, a record in a file of weather observations may have fields recording time, temperature, and

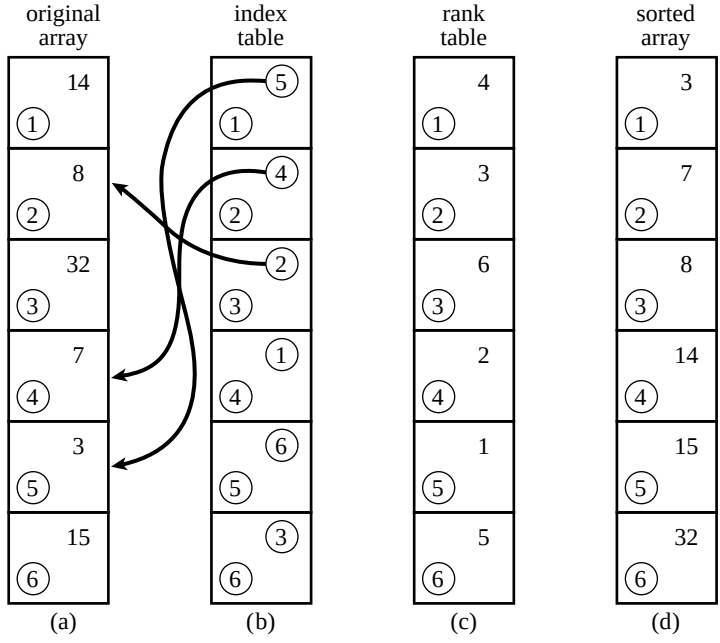


Figure 8.4.1. (a) An unsorted array of six numbers. (b) Index table, whose entries are pointers to the elements of (a) in ascending order. (c) Rank table, whose entries are the ranks of the corresponding elements of (a). (d) Sorted array of the elements in (a).

wind velocity. When we sort the records, we must decide which of these fields we want to be brought into sorted order. The other fields in a record just come along for the ride, and will not, in general, end up in any particular order. The field on which the sort is performed is called the *key* field.

For a data file with many records and many fields, the actual movement of N records into the sorted order of their keys $K_i,\ i = 1, \ldots, N$, can be a daunting task. Instead, one can construct an *index table* $I_j,\ j = 1, \ldots, N$, such that the smallest K_i has $i = I_1$, the second smallest has $i = I_2$, and so on up to the largest K_i with $i = I_N$. In other words, the array

$$K_{I_j} \quad j = 1, 2, \ldots, N \tag{8.4.1}$$

is in sorted order when indexed by j. When an index table is available, one need not move records from their original order. Further, different index tables can be made from the same set of records, indexing them to different keys.

The algorithm for constructing an index table is straightforward: Initialize the index array with the integers from 1 to N, then perform the Quicksort algorithm, moving the elements around *as if* one were sorting the keys. The integer that initially numbered the smallest key thus ends up in the number one position, and so on.

```
SUBROUTINE indexx(n,arr,indx)
INTEGER n,indx(n),M,NSTACK
REAL arr(n)
PARAMETER (M=7,NSTACK=50)
```

Indexes an array `arr(1:n)`, i.e., outputs the array `indx(1:n)` such that `arr(indx(j))` is in ascending order for `j` $= 1, 2, \ldots, N$. The input quantities `n` and `arr` are not changed.

```
      INTEGER i,indxt,ir,itemp,j,jstack,k,l,istack(NSTACK)
      REAL a
      do 11 j=1,n
          indx(j)=j
      enddo 11
      jstack=0
      l=1
      ir=n
1     if(ir-l.lt.M)then
          do 13 j=l+1,ir
              indxt=indx(j)
              a=arr(indxt)
              do 12 i=j-1,l,-1
                  if(arr(indx(i)).le.a)goto 2
                  indx(i+1)=indx(i)
              enddo 12
              i=l-1
2             indx(i+1)=indxt
          enddo 13
          if(jstack.eq.0)return
          ir=istack(jstack)
          l=istack(jstack-1)
          jstack=jstack-2
      else
          k=(l+ir)/2
          itemp=indx(k)
          indx(k)=indx(l+1)
          indx(l+1)=itemp
          if(arr(indx(l)).gt.arr(indx(ir)))then
              itemp=indx(l)
              indx(l)=indx(ir)
              indx(ir)=itemp
          endif
          if(arr(indx(l+1)).gt.arr(indx(ir)))then
              itemp=indx(l+1)
              indx(l+1)=indx(ir)
              indx(ir)=itemp
          endif
          if(arr(indx(l)).gt.arr(indx(l+1)))then
              itemp=indx(l)
              indx(l)=indx(l+1)
              indx(l+1)=itemp
          endif
          i=l+1
          j=ir
          indxt=indx(l+1)
          a=arr(indxt)
3         continue
              i=i+1
          if(arr(indx(i)).lt.a)goto 3
4         continue
              j=j-1
          if(arr(indx(j)).gt.a)goto 4
          if(j.lt.i)goto 5
          itemp=indx(i)
          indx(i)=indx(j)
          indx(j)=itemp
          goto 3
5         indx(l+1)=indx(j)
          indx(j)=indxt
          jstack=jstack+2
          if(jstack.gt.NSTACK)pause 'NSTACK too small in indexx'
          if(ir-i+1.ge.j-l)then
              istack(jstack)=ir
```

```
            istack(jstack-1)=i
            ir=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        endif
    endif
    goto 1
    END
```

If you want to sort an array while making the corresponding rearrangement of several or many other arrays, you should first make an index table, then use it to rearrange each array in turn. This requires two arrays of working space: one to hold the index, and another into which an array is temporarily moved, and from which it is redeposited back on itself in the rearranged order. For 3 arrays, the procedure looks like this:

```
      SUBROUTINE sort3(n,ra,rb,rc,wksp,iwksp)
      INTEGER n,iwksp(n)
      REAL ra(n),rb(n),rc(n),wksp(n)
C     USES indexx
        Sorts an array ra(1:n) into ascending numerical order while making the corresponding
        rearrangements of the arrays rb(1:n) and rc(1:n). An index table is constructed via the
        routine indexx.
      INTEGER j
      call indexx(n,ra,iwksp)          Make the index table.
      do 11 j=1,n                      Save the array ra.
          wksp(j)=ra(j)
      enddo 11
      do 12 j=1,n                      Copy it back in the rearranged order.
          ra(j)=wksp(iwksp(j))
      enddo 12
      do 13 j=1,n                      Ditto rb.
          wksp(j)=rb(j)
      enddo 13
      do 14 j=1,n
          rb(j)=wksp(iwksp(j))
      enddo 14
      do 15 j=1,n                      Ditto rc.
          wksp(j)=rc(j)
      enddo 15
      do 16 j=1,n
          rc(j)=wksp(iwksp(j))
      enddo 16
      return
      END
```

The generalization to any other number of arrays is obviously straightforward.

A *rank table* is different from an index table. A rank table's jth entry gives the rank of the jth element of the original array of keys, ranging from 1 (if that element was the smallest) to N (if that element was the largest). One can easily construct a rank table from an index table, however:

```
SUBROUTINE rank(n,indx,irank)
INTEGER n,indx(n),irank(n)
    Given indx(1:n) as output from the routine indexx, this routine returns an array irank(1:n),
    the corresponding table of ranks.
INTEGER j
do 11 j=1,n
    irank(indx(j))=j
enddo 11
return
END
```

Figure 8.4.1 summarizes the concepts discussed in this section.

8.5 Selecting the Mth Largest

Selection is sorting's austere sister. (Say *that* five times quickly!) Where sorting demands the rearrangement of an entire data array, selection politely asks for a single returned value: What is the kth smallest (or, equivalently, the $m = N+1-k$th largest) element out of N elements? The fastest methods for selection do, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements to the left of the kth, all larger elements to the right, and scrambling the order within each subset. This side effect is at best innocuous, at worst downright inconvenient. When the array is very long, so that making a scratch copy of it is taxing on memory, or when the computational burden of the selection is a negligible part of a larger calculation, one turns to selection algorithms without side effects, which leave the original array undisturbed. Such *in place* selection is slower than the faster selection methods by a factor of about 10. We give routines of both types, below.

The most common use of selection is in the statistical characterization of a set of data. One often wants to know the median element in an array, or the top and bottom quartile elements. When N is odd, the median is the kth element, with $k = (N+1)/2$. When N is even, statistics books define the median as the arithmetic mean of the elements $k = N/2$ and $k = N/2 + 1$ (that is, $N/2$ from the bottom and $N/2$ from the top). If you accept such pedantry, you must perform two separate selections to find these elements. For $N > 100$ we usually define $k = N/2$ to be the median element, pedants be damned.

The fastest general method for selection, allowing rearrangement, is *partitioning*, exactly as was done in the Quicksort algorithm (§8.2). Selecting a "random" partition element, one marches through the array, forcing smaller elements to the left, larger elements to the right. As in Quicksort, it is important to optimize the inner loop, using "sentinels" (§8.2) to minimize the number of comparisons. For sorting, one would then proceed to further partition both subsets. For selection, we can ignore one subset and attend only to the one that contains our desired kth element. Selection by partitioning thus does not need a stack of pending operations, and its operations count scales as N rather than as $N \log N$ (see [1]). Comparison with `sort` in §8.2 should make the following routine obvious:

```
SUBROUTINE rank(n,indx,irank)
INTEGER n,indx(n),irank(n)
    Given indx(1:n) as output from the routine indexx, this routine returns an array irank(1:n),
    the corresponding table of ranks.
INTEGER j
do 11 j=1,n
    irank(indx(j))=j
enddo 11
return
END
```

Figure 8.4.1 summarizes the concepts discussed in this section.

8.5 Selecting the Mth Largest

Selection is sorting's austere sister. (Say *that* five times quickly!) Where sorting demands the rearrangement of an entire data array, selection politely asks for a single returned value: What is the kth smallest (or, equivalently, the $m = N+1-k$th largest) element out of N elements? The fastest methods for selection do, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements to the left of the kth, all larger elements to the right, and scrambling the order within each subset. This side effect is at best innocuous, at worst downright inconvenient. When the array is very long, so that making a scratch copy of it is taxing on memory, or when the computational burden of the selection is a negligible part of a larger calculation, one turns to selection algorithms without side effects, which leave the original array undisturbed. Such *in place* selection is slower than the faster selection methods by a factor of about 10. We give routines of both types, below.

The most common use of selection is in the statistical characterization of a set of data. One often wants to know the median element in an array, or the top and bottom quartile elements. When N is odd, the median is the kth element, with $k = (N+1)/2$. When N is even, statistics books define the median as the arithmetic mean of the elements $k = N/2$ and $k = N/2 + 1$ (that is, $N/2$ from the bottom and $N/2$ from the top). If you accept such pedantry, you must perform two separate selections to find these elements. For $N > 100$ we usually define $k = N/2$ to be the median element, pedants be damned.

The fastest general method for selection, allowing rearrangement, is *partitioning*, exactly as was done in the Quicksort algorithm (§8.2). Selecting a "random" partition element, one marches through the array, forcing smaller elements to the left, larger elements to the right. As in Quicksort, it is important to optimize the inner loop, using "sentinels" (§8.2) to minimize the number of comparisons. For sorting, one would then proceed to further partition both subsets. For selection, we can ignore one subset and attend only to the one that contains our desired kth element. Selection by partitioning thus does not need a stack of pending operations, and its operations count scales as N rather than as $N \log N$ (see [1]). Comparison with `sort` in §8.2 should make the following routine obvious:

```
      FUNCTION select(k,n,arr)
      INTEGER k,n
      REAL select,arr(n)
        Returns the kth smallest value in the array arr(1:n). The input array will be rearranged
        to have this value in location arr(k), with all smaller elements moved to arr(1:k-1) (in
        arbitrary order) and all larger elements in arr[k+1..n] (also in arbitrary order).
      INTEGER i,ir,j,l,mid
      REAL a,temp
      l=1
      ir=n
1     if(ir-l.le.1)then                 Active partition contains 1 or 2 elements.
        if(ir-l.eq.1)then               Active partition contains 2 elements.
          if(arr(ir).lt.arr(l))then
            temp=arr(l)
            arr(l)=arr(ir)
            arr(ir)=temp
          endif
        endif
        select=arr(k)
        return
      else
        mid=(l+ir)/2                    Choose median of left, center, and right elements as par-
        temp=arr(mid)                     titioning element a. Also rearrange so that arr(l) ≤
        arr(mid)=arr(l+1)                 arr(l+1), arr(ir) ≥ arr(l+1).
        arr(l+1)=temp
        if(arr(l).gt.arr(ir))then
          temp=arr(l)
          arr(l)=arr(ir)
          arr(ir)=temp
        endif
        if(arr(l+1).gt.arr(ir))then
          temp=arr(l+1)
          arr(l+1)=arr(ir)
          arr(ir)=temp
        endif
        if(arr(l).gt.arr(l+1))then
          temp=arr(l)
          arr(l)=arr(l+1)
          arr(l+1)=temp
        endif
        i=l+1                           Initialize pointers for partitioning.
        j=ir
        a=arr(l+1)                      Partitioning element.
3       continue                        Beginning of innermost loop.
          i=i+1                         Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
          j=j-1                         Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5                Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                     Exchange elements.
        arr(i)=arr(j)
        arr(j)=temp
        goto 3                          End of innermost loop.
5       arr(l+1)=arr(j)                 Insert partitioning element.
        arr(j)=a
        if(j.ge.k)ir=j-1                Keep active the partition that contains the kth element.
        if(j.le.k)l=i
      endif
      goto 1
      END
```

In-place, nondestructive, selection is conceptually simple, but it requires a lot of bookkeeping, and it is correspondingly slower. The general idea is to pick some number M of elements at random, to sort them, and then to make a pass through the array *counting* how many elements fall in each of the $M + 1$ intervals defined by these elements. The kth largest will fall in one such interval — call it the "live" interval. One then does a second round, first picking M random elements in the live interval, and then determining which of the new, finer, $M + 1$ intervals all presently live elements fall into. And so on, until the kth element is finally localized within a single array of size M, at which point direct selection is possible.

How shall we pick M? The number of rounds, $\log_M N = \log_2 N / \log_2 M$, will be smaller if M is larger; but the work to locate each element among $M + 1$ subintervals will be larger, scaling as $\log_2 M$ for bisection, say. Each round requires looking at all N elements, if only to find those that are still alive, while the bisections are dominated by the N that occur in the first round. Minimizing $O(N \log_M N) + O(N \log_2 M)$ thus yields the result

$$M \sim 2^{\sqrt{\log_2 N}} \tag{8.5.1}$$

The square root of the logarithm is so slowly varying that secondary considerations of machine timing become important. We use $M = 64$ as a convenient constant value.

Two minor additional tricks in the following routine, `selip`, are (i) augmenting the set of M random values by an $M + 1$st, the arithmetic mean, and (ii) choosing the M random values "on the fly" in a pass through the data, by a method that makes later values no less likely to be chosen than earlier ones. (The underlying idea is to give element $m > M$ an M/m chance of being brought into the set. You can prove by induction that this yields the desired result.)

```
      FUNCTION selip(k,n,arr)
      INTEGER k,n,M
      REAL selip,arr(n),BIG
      PARAMETER (M=64,BIG=1.E30)
        Returns the kth smallest value in the array arr(1:n). The input array is not altered.
C     USES shell
      INTEGER i,j,jl,jm,ju,kk,mm,nlo,nxtmm,isel(M+2)
      REAL ahi,alo,sum,sel(M+2)
      if(k.lt.1.or.k.gt.n.or.n.le.0) pause 'bad input to selip'
      kk=k
      ahi=BIG
      alo=-BIG
1     continue                          Main iteration loop, until desired element is isolated.
        mm=0
        nlo=0
        sum=0.
        nxtmm=M+1
        do 11 i=1,n                     Make a pass through the whole array.
          if(arr(i).ge.alo.and.arr(i).le.ahi)then   Consider only elements in the current brackets.
            mm=mm+1
            if(arr(i).eq.alo) nlo=nlo+1             In case of ties for low bracket.
            if(mm.le.M)then             Statistical procedure for selecting m in-range elements
              sel(mm)=arr(i)              with equal probability, even without knowing in
            else if(mm.eq.nxtmm)then      advance how many there are!
              nxtmm=mm+mm/M
              sel(1+mod(i+mm+kk,M))=arr(i)          The mod function provides a somewhat random number.
            endif
            sum=sum+arr(i)
```

```
        endif
      enddo 11
      if(kk.le.nlo)then                 Desired element is tied for lower bound; return it.
        selip=alo
        return
      else if(mm.le.M)then              All in-range elements were kept. So return answer by
        call shell(mm,sel)                  direct method.
        selip=sel(kk)
        return
      endif                             Augment selected set by mean value (fixes degenera-
      sel(M+1)=sum/mm                       cies), and sort it.
      call shell(M+1,sel)
      sel(M+2)=ahi
      do 12 j=1,M+2                     Zero the count array.
        isel(j)=0
      enddo 12
      do 13 i=1,n                       Make another pass through the whole array.
        if(arr(i).ge.alo.and.arr(i).le.ahi)then     For each in-range element..
          jl=0
          ju=M+2
2         if(ju-jl.gt.1)then            ...find its position among the select by bisection...
            jm=(ju+jl)/2
            if(arr(i).ge.sel(jm))then
              jl=jm
            else
              ju=jm
            endif
          goto 2
          endif
          isel(ju)=isel(ju)+1           ...and increment the counter.
        endif
      enddo 13
      j=1                               Now we can narrow the bounds to just one bin, that
3     if(kk.gt.isel(j))then                 is, by a factor of order m.
        alo=sel(j)
        kk=kk-isel(j)
        j=j+1
      goto 3
      endif
      ahi=sel(j)
    goto 1
    END
```

Approximate timings: `selip` is about 10 times slower than `select`. Indeed, for N in the range of $\sim 10^5$, `selip` is about 1.5 times slower than a full sort with `sort`, while `select` is about 6 times faster than `sort`. You should weigh time against memory and convenience carefully.

Of course neither of the above routines should be used for the trivial cases of finding the largest, or smallest, element in an array. Those cases, you code by hand as simple `do` loops. There are also good ways to code the case where k is modest in comparison to N, so that extra memory of order k is not burdensome. An example is to use the method of Heapsort (§8.3) to make a single pass through an array of length N while saving the m *largest* elements. The advantage of the heap structure is that only $\log m$, rather than m, comparisons are required every time a new element is added to the candidate list. This becomes a real savings when $m > O(\sqrt{N})$, but it never hurts otherwise and is easy to code. The following program gives the idea.

```
      SUBROUTINE hpsel(m,n,arr,heap)
      INTEGER m,n
```

```
      REAL arr(n),heap(m)
C     USES sort
        Returns in heap(1:m) the largest m elements of the array arr(1:n), with heap(1) guaranteed to be the the mth largest element. The array arr is not altered. For efficiency, this routine should be used only when m ≪ n.
      INTEGER i,j,k
      REAL swap
      if (m.gt.n/2.or.m.lt.1) pause 'probable misuse of hpsel'
      do 11 i=1,m
          heap(i)=arr(i)
      enddo 11
      call sort(m,heap)                 Create initial heap by overkill! We assume m ≪ n.
      do 12 i=m+1,n                     For each remaining element...
          if(arr(i).gt.heap(1))then     Put it on the heap?
              heap(1)=arr(i)
              j=1
1             continue                  Sift down.
                  k=2*j
                  if(k.gt.m)goto 2
                  if(k.ne.m)then
                      if(heap(k).gt.heap(k+1))k=k+1
                  endif
                  if(heap(j).le.heap(k))goto 2
                  swap=heap(k)
                  heap(k)=heap(j)
                  heap(j)=swap
                  j=k
              goto 1
2             continue
          endif
      enddo 12
      return
      end
```

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).

8.6 Determination of Equivalence Classes

A number of techniques for sorting and searching relate to data structures whose details are beyond the scope of this book, for example, trees, linked lists, etc. These structures and their manipulations are the bread and butter of computer science, as distinct from numerical analysis, and there is no shortage of books on the subject.

In working with experimental data, we have found that one particular such manipulation, namely the determination of equivalence classes, arises sufficiently often to justify inclusion here.

The problem is this: There are N "elements" (or "data points" or whatever), numbered $1, \ldots, N$. You are given pairwise information about whether elements are in the same *equivalence class* of "sameness," by whatever criterion happens to be of interest. For example, you may have a list of facts like: "Element 3 and element 7 are in the same class; element 19 and element 4 are in the same class; element 7 and element 12 are in the same class, ..." Alternatively, you may have a procedure, given the numbers of two elements

```
      REAL arr(n),heap(m)
C     USES sort
        Returns in heap(1:m) the largest m elements of the array arr(1:n), with heap(1) guaranteed to be the the mth largest element. The array arr is not altered. For efficiency, this routine should be used only when m ≪ n.
      INTEGER i,j,k
      REAL swap
      if (m.gt.n/2.or.m.lt.1) pause 'probable misuse of hpsel'
      do 11 i=1,m
          heap(i)=arr(i)
      enddo 11
      call sort(m,heap)                 Create initial heap by overkill! We assume m ≪ n.
      do 12 i=m+1,n                     For each remaining element...
          if(arr(i).gt.heap(1))then     Put it on the heap?
              heap(1)=arr(i)
              j=1
1             continue                  Sift down.
                  k=2*j
                  if(k.gt.m)goto 2
                  if(k.ne.m)then
                      if(heap(k).gt.heap(k+1))k=k+1
                  endif
                  if(heap(j).le.heap(k))goto 2
                  swap=heap(k)
                  heap(k)=heap(j)
                  heap(j)=swap
                  j=k
              goto 1
2             continue
          endif
      enddo 12
      return
      end
```

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).

8.6 Determination of Equivalence Classes

A number of techniques for sorting and searching relate to data structures whose details are beyond the scope of this book, for example, trees, linked lists, etc. These structures and their manipulations are the bread and butter of computer science, as distinct from numerical analysis, and there is no shortage of books on the subject.

In working with experimental data, we have found that one particular such manipulation, namely the determination of equivalence classes, arises sufficiently often to justify inclusion here.

The problem is this: There are N "elements" (or "data points" or whatever), numbered $1, \ldots, N$. You are given pairwise information about whether elements are in the same *equivalence class* of "sameness," by whatever criterion happens to be of interest. For example, you may have a list of facts like: "Element 3 and element 7 are in the same class; element 19 and element 4 are in the same class; element 7 and element 12 are in the same class, ..." Alternatively, you may have a procedure, given the numbers of two elements

j and k, for deciding whether they are in the same class or different classes. (Recall that an equivalence relation can be anything satisfying the *RST properties*: reflexive, symmetric, transitive. This is compatible with any intuitive definition of "sameness.")

The desired output is an assignment to each of the N elements of an equivalence class number, such that two elements are in the same class if and only if they are assigned the same class number.

Efficient algorithms work like this: Let $F(j)$ be the class or "family" number of element j. Start off with each element in its own family, so that $F(j) = j$. The array $F(j)$ can be interpreted as a tree structure, where $F(j)$ denotes the parent of j. If we arrange for each family to be its own tree, disjoint from all the other "family trees," then we can label each family (equivalence class) by its most senior great-great-. . .grandparent. The detailed topology of the tree doesn't matter at all, as long as we graft each related element onto it *somewhere*.

Therefore, we process each elemental datum "j is equivalent to k" by (i) tracking j up to its highest ancestor, (ii) tracking k up to its highest ancestor, (iii) giving j to k as a new parent, or vice versa (it makes no difference). After processing all the relations, we go through all the elements j and reset their $F(j)$'s to their highest possible ancestors, which then label the equivalence classes.

The following routine, based on Knuth [1], assumes that there are `m` elemental pieces of information, stored in two arrays of length `m`, `lista,listb`, the interpretation being that `lista(j)` and `listb(j)`, `j=1...m`, are the numbers of two elements which (we are thus told) are related.

```
      SUBROUTINE eclass(nf,n,lista,listb,m)
      INTEGER m,n,lista(m),listb(m),nf(n)
          Given m equivalences between pairs of n individual elements in the form of the input arrays
          lista(1:m) and listb(1:m), this routine returns in nf(1:n) the number of the equiva-
          lence class of each of the n elements, integers between 1 and n (not all such integers used).
      INTEGER j,k,l
      do 11 k=1,n                       Initialize each element its own class.
          nf(k)=k
      enddo 11
      do 12 l=1,m                       For each piece of input information...
          j=lista(l)
1         if(nf(j).ne.j)then            Track first element up to its ancestor.
              j=nf(j)
          goto 1
          endif
          k=listb(l)
2         if(nf(k).ne.k)then            Track second element up to its ancestor.
              k=nf(k)
          goto 2
          endif
          if(j.ne.k)nf(j)=k             If they are not already related, make them so.
      enddo 12
      do 13 j=1,n                       Final sweep up to highest ancestors.
3         if(nf(j).ne.nf(nf(j)))then
              nf(j)=nf(nf(j))
          goto 3
          endif
      enddo 13
      return
      END
```

Alternatively, we may be able to construct a procedure `equiv(j,k)` that returns a value `.true.` if elements `j` and `k` are related, or `.false.` if they are not. Then we want to loop over all pairs of elements to get the complete picture. D. Eardley has devised a clever way of doing this while simultaneously sweeping the tree up to high ancestors in a manner that keeps it current and obviates most of the final sweep phase:

```
SUBROUTINE eclazz(nf,n,equiv)
INTEGER n,nf(n)
LOGICAL equiv
EXTERNAL equiv
    Given a user-supplied logical function equiv which tells whether a pair of elements, each
    in the range 1...n, are related, return in nf equivalence class numbers for each element.
INTEGER jj,kk
nf(1)=1
do 12 jj=2,n                    Loop over first element of all pairs.
    nf(jj)=jj
    do 11 kk=1,jj-1             Loop over second element of all pairs.
        nf(kk)=nf(nf(kk))       Sweep it up this much.
        if (equiv(jj,kk)) nf(nf(nf(kk)))=jj     Good exercise for the reader to figure
    enddo 11                                        out why this much ancestry is
enddo 12                                            necessary!
do 13 jj=1,n                    Only this much sweeping is needed finally.
    nf(jj)=nf(nf(jj))
enddo 13
return
END
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.3.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 30.

Chapter 9. Root Finding and Nonlinear Sets of Equations

9.0 Introduction

We now consider that most basic of tasks, solving equations numerically. While most equations are born with both a right-hand side and a left-hand side, one traditionally moves all terms to the left, leaving

$$f(x) = 0 \tag{9.0.1}$$

whose solution or solutions are desired. When there is only one independent variable, the problem is *one-dimensional*, namely to find the root or roots of a function.

With more than one independent variable, more than one equation can be satisfied simultaneously. You likely once learned the *implicit function theorem* which (in this context) gives us the hope of satisfying N equations in N unknowns simultaneously. Note that we have only hope, not certainty. A nonlinear set of equations may have no (real) solutions at all. Contrariwise, it may have more than one solution. The implicit function theorem tells us that "generically" the solutions will be distinct, pointlike, and separated from each other. If, however, life is so unkind as to present you with a nongeneric, i.e., degenerate, case, then you can get a continuous family of solutions. In vector notation, we want to find one or more N-dimensional solution vectors $\mathbf{x}$ such that

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{9.0.2}$$

where $\mathbf{f}$ is the N-dimensional vector-valued function whose components are the individual equations to be satisfied simultaneously.

Don't be fooled by the apparent notational similarity of equations (9.0.2) and (9.0.1). Simultaneous solution of equations in N dimensions is *much* more difficult than finding roots in the one-dimensional case. The principal difference between one and many dimensions is that, in one dimension, it is possible to bracket or "trap" a root between bracketing values, and then hunt it down like a rabbit. In multidimensions, you can never be sure that the root is there at all until you have found it.

Except in linear problems, root finding invariably proceeds by iteration, and this is equally true in one or in many dimensions. Starting from some approximate trial solution, a useful algorithm will improve the solution until some predetermined convergence criterion is satisfied. For smoothly varying functions, good algorithms

will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming's motto, "the purpose of computing is insight, not numbers," is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with ten-digit accuracy, to the wrong root of a problem, or whenever it fails to converge because there is actually *no* root, or because there is a root but your initial estimate was not sufficiently close to it.

"This talk of insight is all very well, but what do I actually do?" For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root, that is, know that the function changes sign in an identified interval, before trying to converge to the root's value.

Finally (this is advice with which some daring souls might disagree, but we give it nonetheless) never let your iteration method get outside of the best bracketing bounds obtained at any stage. We will see below that some pedagogically important algorithms, such as *secant method* or *Newton-Raphson*, can violate this last constraint, and are thus not recommended unless certain fixups are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number. In that case, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-liners: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 10 to determine whether a tantalizing dip in the function really does cross zero or not. (You can easily modify the simple golden section routine of §10.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a *double* root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine `rtsafe` in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned!
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works *very* well if you can supply a

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally steer clear of interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively scaled axes, can save you a lot of grief as you enter the world of root finding.

```
      SUBROUTINE scrsho(fx)
      INTEGER ISCR,JSCR
      REAL fx
      EXTERNAL fx
      PARAMETER (ISCR=60,JSCR=21)      Number of horizontal and vertical positions in display.
        For interactive CRT terminal use. Produce a crude graph of the function fx over the
        prompted-for interval x1,x2. Query for another plot until the user signals satisfaction.
      INTEGER i,j,jz
      REAL dx,dyj,x,x1,x2,ybig,ysml,y(ISCR)
      CHARACTER*1 scr(ISCR,JSCR),blank,zero,yy,xx,ff
      SAVE blank,zero,yy,xx,ff
      DATA blank,zero,yy,xx,ff/' ','-','l','-','x'/
1     continue
      write (*,*) ' Enter x1,x2 (= to stop)' Query for another plot, quit if x1=x2.
      read (*,*) x1,x2
      if(x1.eq.x2) return
      do 11 j=1,JSCR                   Fill vertical sides with character 'l'.
        scr(1,j)=yy
        scr(ISCR,j)=yy
      enddo 11
      do 13 i=2,ISCR-1
        scr(i,1)=xx                    Fill top, bottom with character '-'.
        scr(i,JSCR)=xx
        do 12 j=2,JSCR-1               Fill interior with blanks.
          scr(i,j)=blank
        enddo 12
      enddo 13
      dx=(x2-x1)/(ISCR-1)
      x=x1
      ybig=0.                          Limits will include 0.
      ysml=ybig
      do 14 i=1,ISCR                   Evaluate the function at equal intervals. Find the
        y(i)=fx(x)                       largest and smallest values.
        if(y(i).lt.ysml) ysml=y(i)
        if(y(i).gt.ybig) ybig=y(i)
        x=x+dx
      enddo 14
      if(ybig.eq.ysml) ybig=ysml+1.    Be sure to separate top and bottom.
      dyj=(JSCR-1)/(ybig-ysml)
      jz=1-ysml*dyj                    Note which row corresponds to 0.
      do 15 i=1,ISCR                   Place an indicator at function height and 0.
        scr(i,jz)=zero
        j=1+(y(i)-ysml)*dyj
        scr(i,j)=ff
      enddo 15
      write (*,'(1x,1pe10.3,1x,80a1)') ybig,(scr(i,JSCR),i=1,ISCR)
      do 16 j=JSCR-1,2,-1              Display.
        write (*,'(12x,80a1)') (scr(i,j),i=1,ISCR)
      enddo 16
      write (*,'(1x,1pe10.3,1x,80a1)') ysml,(scr(i,1),i=1,ISCR)
      write (*,'(12x,1pe10.3,40x,e10.3)') x1,x2
      goto 1
      END
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 5.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapters 2, 7, and 14.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 8.

Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill).

9.1 Bracketing and Bisection

We will say that a root is *bracketed* in the interval (a, b) if $f(a)$ and $f(b)$ have opposite signs. If the function is continuous, then at least one root must lie in that interval (the *intermediate value theorem*). If the function is discontinuous, but bounded, then instead of a root there might be a step discontinuity which crosses zero (see Figure 9.1.1). For numerical purposes, that might as well be a root, since the behavior is indistinguishable from the case of a continuous function whose zero crossing occurs in between two "adjacent" floating-point numbers in a machine's finite-precision representation. Only for functions with singularities is there the possibility that a bracketed root is not really there, as for example

$$f(x) = \frac{1}{x - c} \tag{9.1.1}$$

Some root-finding algorithms (e.g., bisection in this section) will readily converge to c in (9.1.1). Luckily there is not much possibility of your mistaking c, or any number x close to it, for a root, since mere evaluation of $|f(x)|$ will give a very large, rather than a very small, result.

If you are given a function in a black box, there is no sure way of bracketing its roots, or of even determining that it has roots. If you like pathological examples, think about the problem of locating the two real roots of equation (3.0.1), which dips below zero only in the ridiculously small interval of about $x = \pi \pm 10^{-667}$.

In the next chapter we will deal with the related problem of bracketing a function's minimum. There it is possible to give a procedure that always succeeds; in essence, "Go downhill, taking steps of increasing size, until your function starts back uphill." There is no analogous procedure for roots. The procedure "go downhill until your function changes sign," can be foiled by a function that has a simple extremum. Nevertheless, if you are prepared to deal with a "failure" outcome, this procedure is often a good first start; success is usual if your function has opposite signs in the limit $x \to \pm\infty$.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 5.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapters 2, 7, and 14.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 8.

Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill).

9.1 Bracketing and Bisection

We will say that a root is *bracketed* in the interval (a, b) if $f(a)$ and $f(b)$ have opposite signs. If the function is continuous, then at least one root must lie in that interval (the *intermediate value theorem*). If the function is discontinuous, but bounded, then instead of a root there might be a step discontinuity which crosses zero (see Figure 9.1.1). For numerical purposes, that might as well be a root, since the behavior is indistinguishable from the case of a continuous function whose zero crossing occurs in between two "adjacent" floating-point numbers in a machine's finite-precision representation. Only for functions with singularities is there the possibility that a bracketed root is not really there, as for example

$$f(x) = \frac{1}{x - c} \tag{9.1.1}$$

Some root-finding algorithms (e.g., bisection in this section) will readily converge to c in (9.1.1). Luckily there is not much possibility of your mistaking c, or any number x close to it, for a root, since mere evaluation of $|f(x)|$ will give a very large, rather than a very small, result.

If you are given a function in a black box, there is no sure way of bracketing its roots, or of even determining that it has roots. If you like pathological examples, think about the problem of locating the two real roots of equation (3.0.1), which dips below zero only in the ridiculously small interval of about $x = \pi \pm 10^{-667}$.

In the next chapter we will deal with the related problem of bracketing a function's minimum. There it is possible to give a procedure that always succeeds; in essence, "Go downhill, taking steps of increasing size, until your function starts back uphill." There is no analogous procedure for roots. The procedure "go downhill until your function changes sign," can be foiled by a function that has a simple extremum. Nevertheless, if you are prepared to deal with a "failure" outcome, this procedure is often a good first start; success is usual if your function has opposite signs in the limit $x \to \pm\infty$.

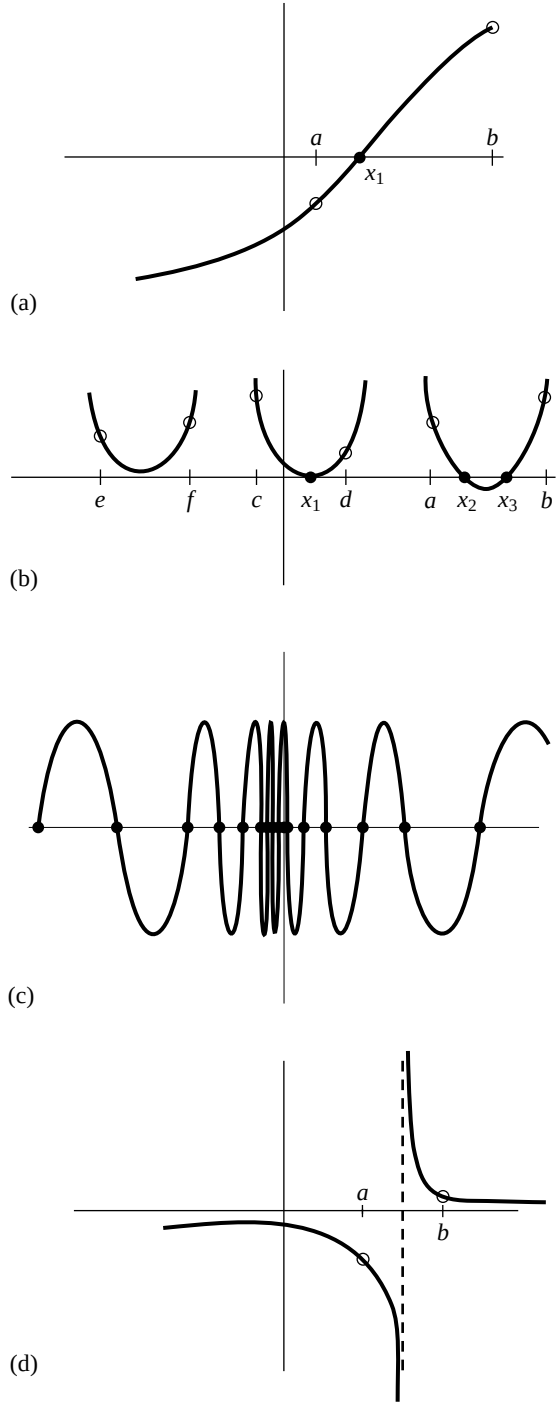


Figure 9.1.1. Some situations encountered while root finding: (a) shows an isolated root x_1 bracketed by two points a and b at which the function has opposite signs; (b) illustrates that there is not necessarily a sign change in the function near a double root (in fact, there is not necessarily a root!); (c) is a pathological function with many roots; in (d) the function has opposite signs at points a and b, but the points bracket a singularity, not a root.

```
SUBROUTINE zbrac(func,x1,x2,succes)
INTEGER NTRY
REAL x1,x2,func,FACTOR
EXTERNAL func
PARAMETER (FACTOR=1.6,NTRY=50)
    Given a function func and an initial guessed range x1 to x2, the routine expands the range
    geometrically until a root is bracketed by the returned values x1 and x2 (in which case
    succes returns as .true.) or until the range becomes unacceptably large (in which case
    succes returns as .false.).
INTEGER j
REAL f1,f2
LOGICAL succes
if(x1.eq.x2)pause 'you have to guess an initial range in zbrac'
f1=func(x1)
f2=func(x2)
succes=.true.
do 11 j=1,NTRY
    if(f1*f2.lt.0.)return
    if(abs(f1).lt.abs(f2))then
        x1=x1+FACTOR*(x1-x2)
        f1=func(x1)
    else
        x2=x2+FACTOR*(x2-x1)
        f2=func(x2)
    endif
enddo 11
succes=.false.
return
END
```

Alternatively, you might want to "look inward" on an initial interval, rather than "look outward" from it, asking if there are any roots of the function $f(x)$ in the interval from x_1 to x_2 when a search is carried out by subdivision into n equal intervals. The following subroutine returns brackets for up to nb distinct intervals which each contain one or more roots.

```
SUBROUTINE zbrak(fx,x1,x2,n,xb1,xb2,nb)
INTEGER n,nb
REAL x1,x2,xb1(nb),xb2(nb),fx
EXTERNAL fx
    Given a function fx defined on the interval from x1-x2 subdivide the interval into n equally
    spaced segments, and search for zero crossings of the function. nb is input as the maxi-
    mum number of roots sought, and is reset to the number of bracketing pairs xb1(1:nb),
    xb2(1:nb) that are found.
INTEGER i,nbb
REAL dx,fc,fp,x
nbb=0
x=x1
dx=(x2-x1)/n                    Determine the spacing appropriate to the mesh.
fp=fx(x)
do 11 i=1,n                     Loop over all intervals
    x=x+dx
    fc=fx(x)
    if(fc*fp.le.0.) then        If a sign change occurs then record values for the bounds.
        nbb=nbb+1
        xb1(nbb)=x-dx
        xb2(nbb)=x
        if(nbb.eq.nb)goto 1
    endif
    fp=fc
enddo 11
```

```
1   continue
    nb=nbb
    return
    END
```

Bisection Method

Once we know that an interval contains a root, several classical procedures are available to refine it. These proceed with varying degrees of speed and sureness towards the answer. Unfortunately, the methods that are guaranteed to converge plod along most slowly, while those that rush to the solution in the best cases can also dash rapidly to infinity without warning if measures are not taken to avoid such behavior.

The *bisection method* is one that cannot fail. It is thus not to be sneered at as a method for otherwise badly behaved problems. The idea is simple. Over some interval the function is known to pass through zero because it changes sign. Evaluate the function at the interval's midpoint and examine its sign. Use the midpoint to replace whichever limit has the same sign. After each iteration the bounds containing the root decrease by a factor of two. If after n iterations the root is known to be within an interval of size ϵ_n, then after the next iteration it will be bracketed within an interval of size

$$\epsilon_{n+1} = \epsilon_n/2 \tag{9.1.2}$$

neither more nor less. Thus, we know in advance the number of iterations required to achieve a given tolerance in the solution,

$$n = \log_2 \frac{\epsilon_0}{\epsilon} \tag{9.1.3}$$

where ϵ_0 is the size of the initially bracketing interval, ϵ is the desired ending tolerance.

Bisection *must* succeed. If the interval happens to contain two or more roots, bisection will find one of them. If the interval contains no roots and merely straddles a singularity, it will converge on the singularity.

When a method converges as a factor (less than 1) times the previous uncertainty to the first power (as is the case for bisection), it is said to converge *linearly*. Methods that converge as a higher power,

$$\epsilon_{n+1} = \text{constant} \times (\epsilon_n)^m \qquad m > 1 \tag{9.1.4}$$

are said to converge superlinearly. In other contexts "linear" convergence would be termed "exponential," or "geometrical." That is not too bad at all: Linear convergence means that successive significant figures are won linearly with computational effort.

It remains to discuss practical criteria for convergence. It is crucial to keep in mind that computers use a fixed number of binary digits to represent floating-point numbers. While your function might analytically pass through zero, it is possible that its computed value is never zero, for any floating-point argument. One must decide what accuracy on the root is attainable: Convergence to within 10^{-6} in absolute value is reasonable when the root lies near 1, but certainly unachievable if

the root lies near 10^{26}. One might thus think to specify convergence by a relative (fractional) criterion, but this becomes unworkable for roots near zero. To be most general, the routines below will require you to specify an absolute tolerance, such that iterations continue until the interval becomes smaller than this tolerance in absolute units. Usually you may wish to take the tolerance to be $\epsilon(|x_1|+|x_2|)/2$ where ϵ is the machine precision and x_1 and x_2 are the initial brackets. When the root lies near zero you ought to consider carefully what reasonable tolerance means for your function. The following routine quits after 40 bisections in any event, with $2^{-40} \approx 10^{-12}$.

```
FUNCTION rtbis(func,x1,x2,xacc)
INTEGER JMAX
REAL rtbis,x1,x2,xacc,func
EXTERNAL func
PARAMETER (JMAX=40)          Maximum allowed number of bisections.
    Using bisection, find the root of a function func known to lie between x1 and x2. The
    root, returned as rtbis, will be refined until its accuracy is ±xacc.
INTEGER j
REAL dx,f,fmid,xmid
fmid=func(x2)
f=func(x1)
if(f*fmid.ge.0.) pause 'root must be bracketed in rtbis'
if(f.lt.0.)then              Orient the search so that f>0 lies at x+dx.
    rtbis=x1
    dx=x2-x1
else
    rtbis=x2
    dx=x1-x2
endif
do 11 j=1,JMAX               Bisection loop.
    dx=dx*.5
    xmid=rtbis+dx
    fmid=func(xmid)
    if(fmid.le.0.)rtbis=xmid
    if(abs(dx).lt.xacc .or. fmid.eq.0.) return
enddo 11
pause 'too many bisections in rtbis'
END
```

9.2 Secant Method, False Position Method, and Ridders' Method

For functions that are smooth near a root, the methods known respectively as *false position* (or *regula falsi*) and *secant method* generally converge faster than bisection. In both of these methods the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis. After each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root.

The *only* difference between the methods is that secant retains the most recent of the prior estimates (Figure 9.2.1; this requires an arbitrary choice on the first iteration), while false position retains that prior estimate for which the function value

the root lies near 10^{26}. One might thus think to specify convergence by a relative (fractional) criterion, but this becomes unworkable for roots near zero. To be most general, the routines below will require you to specify an absolute tolerance, such that iterations continue until the interval becomes smaller than this tolerance in absolute units. Usually you may wish to take the tolerance to be $\epsilon(|x_1|+|x_2|)/2$ where ϵ is the machine precision and x_1 and x_2 are the initial brackets. When the root lies near zero you ought to consider carefully what reasonable tolerance means for your function. The following routine quits after 40 bisections in any event, with $2^{-40} \approx 10^{-12}$.

```
FUNCTION rtbis(func,x1,x2,xacc)
INTEGER JMAX
REAL rtbis,x1,x2,xacc,func
EXTERNAL func
PARAMETER (JMAX=40)          Maximum allowed number of bisections.
    Using bisection, find the root of a function func known to lie between x1 and x2. The
    root, returned as rtbis, will be refined until its accuracy is ±xacc.
INTEGER j
REAL dx,f,fmid,xmid
fmid=func(x2)
f=func(x1)
if(f*fmid.ge.0.) pause 'root must be bracketed in rtbis'
if(f.lt.0.)then              Orient the search so that f>0 lies at x+dx.
    rtbis=x1
    dx=x2-x1
else
    rtbis=x2
    dx=x1-x2
endif
do 11 j=1,JMAX               Bisection loop.
    dx=dx*.5
    xmid=rtbis+dx
    fmid=func(xmid)
    if(fmid.le.0.)rtbis=xmid
    if(abs(dx).lt.xacc .or. fmid.eq.0.) return
enddo 11
pause 'too many bisections in rtbis'
END
```

9.2 Secant Method, False Position Method, and Ridders' Method

For functions that are smooth near a root, the methods known respectively as *false position* (or *regula falsi*) and *secant method* generally converge faster than bisection. In both of these methods the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis. After each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root.

The *only* difference between the methods is that secant retains the most recent of the prior estimates (Figure 9.2.1; this requires an arbitrary choice on the first iteration), while false position retains that prior estimate for which the function value

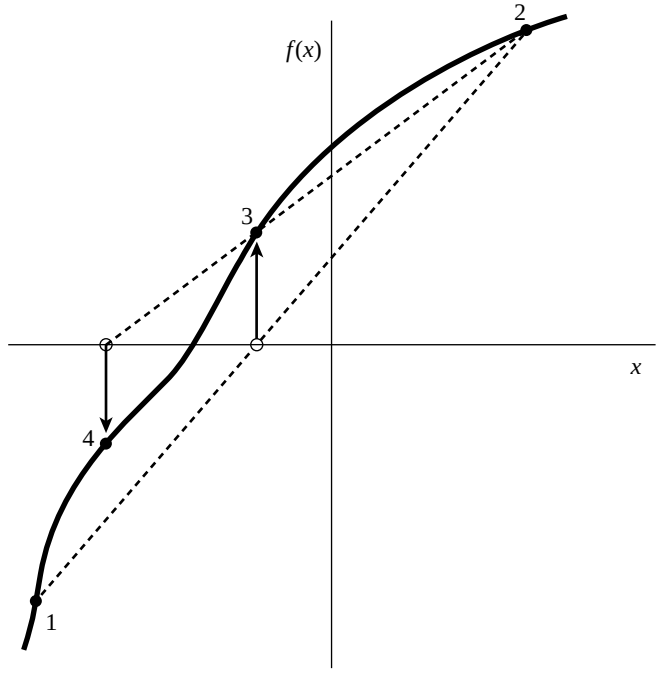


Figure 9.2.1. Secant method. Extrapolation or interpolation lines (dashed) are drawn through the two most recently evaluated points, whether or not they bracket the function. The points are numbered in the order that they are used.

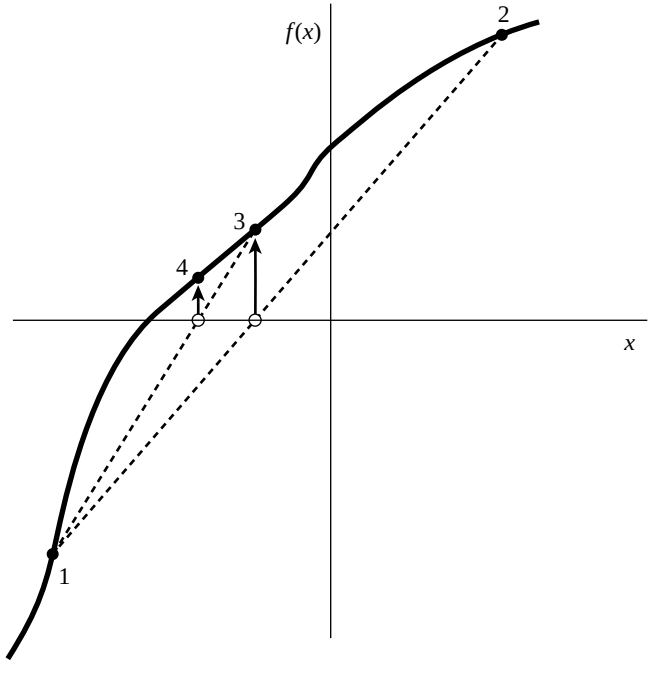


Figure 9.2.2. False position method. Interpolation lines (dashed) are drawn through the most recent points *that bracket the root.* In this example, point 1 thus remains "active" for many steps. False position converges less rapidly than the secant method, but it is more certain.

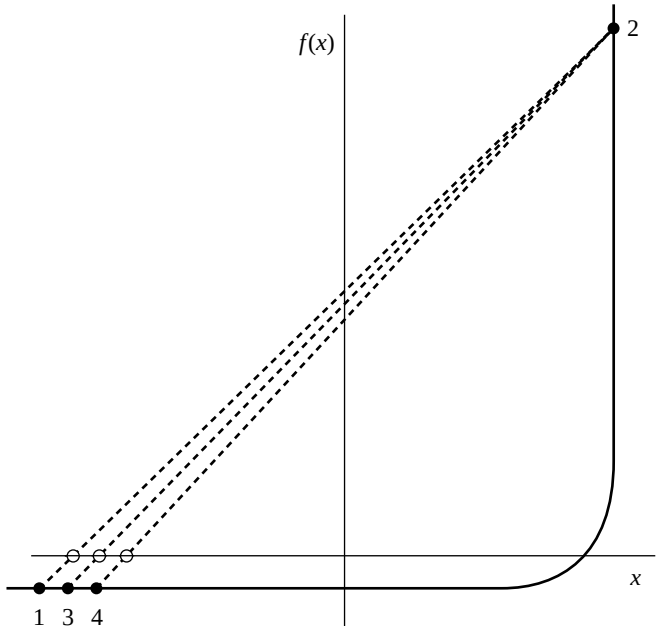


Figure 9.2.3. Example where both the secant and false position methods will take many iterations to arrive at the true root. This function would be difficult for many other root-finding methods.

has opposite sign from the function value at the current best estimate of the root, so that the two points continue to bracket the root (Figure 9.2.2). Mathematically, the secant method converges more rapidly near a root of a sufficiently continuous function. Its order of convergence can be shown to be the "golden ratio" $1.618\ldots$, so that

$$\lim_{k\to\infty} |\epsilon_{k+1}| \approx \text{const} \times |\epsilon_k|^{1.618} \tag{9.2.1}$$

The secant method has, however, the disadvantage that the root does not necessarily remain bracketed. For functions that are *not* sufficiently continuous, the algorithm can therefore not be guaranteed to converge: Local behavior might send it off towards infinity.

False position, since it sometimes keeps an older rather than newer function evaluation, has a lower order of convergence. Since the newer function value will *sometimes* be kept, the method is often superlinear, but estimation of its exact order is not so easy.

Here are sample implementations of these two related methods. While these methods are standard textbook fare, *Ridders' method*, described below, or *Brent's method*, in the next section, are almost always better choices. Figure 9.2.3 shows the behavior of secant and false-position methods in a difficult situation.

```
FUNCTION rtflsp(func,x1,x2,xacc)
INTEGER MAXIT
REAL rtflsp,x1,x2,xacc,func
EXTERNAL func
PARAMETER (MAXIT=30)          Set to the maximum allowed number of iterations.
```

Using the false position method, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `rtflsp`, is refined until its accuracy is $\pm$`xacc`.

```
INTEGER j
REAL del,dx,f,fh,fl,swap,xh,xl
fl=func(x1)
fh=func(x2)                    Be sure the interval brackets a root.
if(fl*fh.gt.0.) pause 'root must be bracketed in rtflsp'
if(fl.lt.0.)then               Identify the limits so that xl corresponds to the low side.
    xl=x1
    xh=x2
else
    xl=x2
    xh=x1
    swap=fl
    fl=fh
    fh=swap
endif
dx=xh-xl
do 11 j=1,MAXIT                False position loop.
    rtflsp=xl+dx*fl/(fl-fh)    Increment with respect to latest value.
    f=func(rtflsp)
    if(f.lt.0.) then           Replace appropriate limit.
        del=xl-rtflsp
        xl=rtflsp
        fl=f
    else
        del=xh-rtflsp
        xh=rtflsp
        fh=f
    endif
    dx=xh-xl
    if(abs(del).lt.xacc.or.f.eq.0.)return        Convergence.
enddo 11
pause 'rtflsp exceed maximum iterations'
END
```

```
FUNCTION rtsec(func,x1,x2,xacc)
INTEGER MAXIT
REAL rtsec,x1,x2,xacc,func
EXTERNAL func
PARAMETER (MAXIT=30)           Maximum allowed number of iterations.
```

Using the secant method, find the root of a function `func` thought to lie between `x1` and `x2`. The root, returned as `rtsec`, is refined until its accuracy is $\pm$`xacc`.

```
INTEGER j
REAL dx,f,fl,swap,xl
fl=func(x1)
f=func(x2)
if(abs(fl).lt.abs(f))then      Pick the bound with the smaller function value as the most
    rtsec=x1                       recent guess.
    xl=x2
    swap=fl
    fl=f
    f=swap
else
    xl=x1
    rtsec=x2
endif
do 11 j=1,MAXIT                Secant loop.
    dx=(xl-rtsec)*f/(f-fl)     Increment with respect to latest value.
    xl=rtsec
    fl=f
    rtsec=rtsec+dx
```

```
    f=func(rtsec)
    if(abs(dx).lt.xacc.or.f.eq.0.)return      Convergence.
enddo 11
pause 'rtsec exceed maximum iterations'
END
```

Ridders' Method

A powerful variant on false position is due to Ridders [1]. When a root is bracketed between x_1 and x_2, Ridders' method first evaluates the function at the midpoint $x_3 = (x_1 + x_2)/2$. It then factors out that unique exponential function which turns the residual function into a straight line. Specifically, it solves for a factor e^Q that gives

$$f(x_1) - 2f(x_3)e^Q + f(x_2)e^{2Q} = 0 \tag{9.2.2}$$

This is a quadratic equation in e^Q, which can be solved to give

$$e^Q = \frac{f(x_3) + \text{sign}[f(x_2)]\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}{f(x_2)} \tag{9.2.3}$$

Now the false position method is applied, not to the values $f(x_1), f(x_3), f(x_2)$, but to the values $f(x_1), f(x_3)e^Q, f(x_2)e^{2Q}$, yielding a new guess for the root, x_4. The overall updating formula (incorporating the solution 9.2.3) is

$$x_4 = x_3 + (x_3 - x_1)\frac{\text{sign}[f(x_1) - f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}} \tag{9.2.4}$$

Equation (9.2.4) has some very nice properties. First, x_4 is guaranteed to lie in the interval (x_1, x_2), so the method never jumps out of its brackets. Second, the convergence of successive applications of equation (9.2.4) is *quadratic*, that is, $m = 2$ in equation (9.1.4). Since each application of (9.2.4) requires two function evaluations, the actual order of the method is $\sqrt{2}$, not 2; but this is still quite respectably superlinear: the number of significant digits in the answer approximately *doubles* with each two function evaluations. Third, taking out the function's "bend" via exponential (that is, ratio) factors, rather than via a polynomial technique (e.g., fitting a parabola), turns out to give an extraordinarily robust algorithm. In both reliability and speed, Ridders' method is generally competitive with the more highly developed and better established (but more complicated) method of Van Wijngaarden, Dekker, and Brent, which we next discuss.

```
      FUNCTION zriddr(func,x1,x2,xacc)
      INTEGER MAXIT
      REAL zriddr,x1,x2,xacc,func,UNUSED
      PARAMETER (MAXIT=60,UNUSED=-1.11E30)
      EXTERNAL func
C     USES func
        Using Ridders' method, return the root of a function func known to lie between x1 and
        x2. The root, returned as zriddr, will be refined to an approximate accuracy xacc.
      INTEGER j
      REAL fh,fl,fm,fnew,s,xh,xl,xm,xnew
```

```
fl=func(x1)
fh=func(x2)
if((fl.gt.0..and.fh.lt.0.).or.(fl.lt.0..and.fh.gt.0.))then
    xl=x1
    xh=x2
    zriddr=UNUSED                          Any highly unlikely value, to simplify logic
    do 11 j=1,MAXIT                            below.
        xm=0.5*(xl+xh)
        fm=func(xm)                        First of two function evaluations per it-
        s=sqrt(fm**2-fl*fh)                    eration.
        if(s.eq.0.)return
        xnew=xm+(xm-xl)*(sign(1.,fl-fh)*fm/s)   Updating formula.
        if (abs(xnew-zriddr).le.xacc) return
        zriddr=xnew
        fnew=func(zriddr)                  Second of two function evaluations per
        if (fnew.eq.0.) return                 iteration.
        if(sign(fm,fnew).ne.fm) then       Bookkeeping to keep the root bracketed
            xl=xm                              on next iteration.
            fl=fm
            xh=zriddr
            fh=fnew
        else if(sign(fl,fnew).ne.fl) then
            xh=zriddr
            fh=fnew
        else if(sign(fh,fnew).ne.fh) then
            xl=zriddr
            fl=fnew
        else
            pause 'never get here in zriddr'
        endif
        if(abs(xh-xl).le.xacc) return
    enddo 11
    pause 'zriddr exceed maximum iterations'
else if (fl.eq.0.) then
    zriddr=x1
else if (fh.eq.0.) then
    zriddr=x2
else
    pause 'root must be bracketed in zriddr'
endif
return
END
```

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.3.

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press), Chapter 12.

Ridders, C.J.F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979–980. [1]

9.3 Van Wijngaarden–Dekker–Brent Method

While secant and false position formally converge faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly.

```
fl=func(x1)
fh=func(x2)
if((fl.gt.0..and.fh.lt.0.).or.(fl.lt.0..and.fh.gt.0.))then
    xl=x1
    xh=x2
    zriddr=UNUSED                          Any highly unlikely value, to simplify logic
    do 11 j=1,MAXIT                            below.
        xm=0.5*(xl+xh)
        fm=func(xm)                        First of two function evaluations per it-
        s=sqrt(fm**2-fl*fh)                    eration.
        if(s.eq.0.)return
        xnew=xm+(xm-xl)*(sign(1.,fl-fh)*fm/s)   Updating formula.
        if (abs(xnew-zriddr).le.xacc) return
        zriddr=xnew
        fnew=func(zriddr)                  Second of two function evaluations per
        if (fnew.eq.0.) return                 iteration.
        if(sign(fm,fnew).ne.fm) then       Bookkeeping to keep the root bracketed
            xl=xm                              on next iteration.
            fl=fm
            xh=zriddr
            fh=fnew
        else if(sign(fl,fnew).ne.fl) then
            xh=zriddr
            fh=fnew
        else if(sign(fh,fnew).ne.fh) then
            xl=zriddr
            fl=fnew
        else
            pause 'never get here in zriddr'
        endif
        if(abs(xh-xl).le.xacc) return
    enddo 11
    pause 'zriddr exceed maximum iterations'
else if (fl.eq.0.) then
    zriddr=x1
else if (fh.eq.0.) then
    zriddr=x2
else
    pause 'root must be bracketed in zriddr'
endif
return
END
```

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.3.

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press), Chapter 12.

Ridders, C.J.F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979–980. [1]

9.3 Van Wijngaarden–Dekker–Brent Method

While secant and false position formally converge faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly.

These can be choppy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while secant and false position can sometimes spend many cycles slowly pulling distant bounds closer to a root. Ridders' method does a much better job, but it too can sometimes be fooled. Is there a way to combine superlinear convergence with the sureness of bisection?

Yes. We can keep track of whether a supposedly superlinear method is actually converging the way it is supposed to, and, if it is not, we can intersperse bisection steps so as to guarantee *at least* linear convergence. This kind of super-strategy requires attention to bookkeeping detail, and also careful consideration of how roundoff errors can affect the guiding strategy. Also, we must be able to determine reliably when convergence has been achieved.

An excellent algorithm that pays close attention to these matters was developed in the 1960s by van Wijngaarden, Dekker, and others at the Mathematical Center in Amsterdam, and later improved by Brent [1]. For brevity, we refer to the final form of the algorithm as *Brent's method*. The method is *guaranteed* (by Brent) to converge, so long as the function can be evaluated within the initial interval known to contain a root.

Brent's method combines root bracketing, bisection, and *inverse quadratic interpolation* to converge from the neighborhood of a zero crossing. While the false position and secant methods assume approximately linear behavior between two prior root estimates, inverse quadratic interpolation uses three prior points to fit an inverse quadratic function (x as a quadratic function of y) whose value at $y = 0$ is taken as the next estimate of the root x. Of course one must have contingency plans for what to do if the root falls outside of the brackets. Brent's method takes care of all that. If the three point pairs are $[a, f(a)], [b, f(b)], [c, f(c)]$ then the interpolation formula (cf. equation 3.1.1) is

$$x = \frac{[y - f(a)][y - f(b)]c}{[f(c) - f(a)][f(c) - f(b)]} + \frac{[y - f(b)][y - f(c)]a}{[f(a) - f(b)][f(a) - f(c)]} + \frac{[y - f(c)][y - f(a)]b}{[f(b) - f(c)][f(b) - f(a)]} \tag{9.3.1}$$

Setting y to zero gives a result for the next root estimate, which can be written as

$$x = b + P/Q \tag{9.3.2}$$

where, in terms of

$$R \equiv f(b)/f(c), \qquad S \equiv f(b)/f(a), \qquad T \equiv f(a)/f(c) \tag{9.3.3}$$

we have

$$P = S\,[T(R - T)(c - b) - (1 - R)(b - a)] \tag{9.3.4}$$

$$Q = (T - 1)(R - 1)(S - 1) \tag{9.3.5}$$

In practice b is the current best estimate of the root and P/Q ought to be a "small" correction. Quadratic methods work well only when the function behaves smoothly;

they run the serious risk of giving very bad estimates of the next root or causing machine failure by an inappropriate division by a very small number ($Q \approx 0$). Brent's method guards against this problem by maintaining brackets on the root and checking where the interpolation would land before carrying out the division. When the correction P/Q would not land within the bounds, or when the bounds are not collapsing rapidly enough, the algorithm takes a bisection step. Thus, Brent's method combines the sureness of bisection with the speed of a higher-order method when appropriate. We recommend it as the method of choice for general one-dimensional root finding where a function's values only (and not its derivative or functional form) are available.

```
      FUNCTION zbrent(func,x1,x2,tol)
      INTEGER ITMAX
      REAL zbrent,tol,x1,x2,func,EPS
      EXTERNAL func
      PARAMETER (ITMAX=100,EPS=3.e-8)
        Using Brent's method, find the root of a function func known to lie between x1 and x2.
        The root, returned as zbrent, will be refined until its accuracy is tol.
        Parameters: Maximum allowed number of iterations, and machine floating-point precision.
      INTEGER iter
      REAL a,b,c,d,e,fa,fb,fc,p,q,r,
     *     s,tol1,xm
      a=x1
      b=x2
      fa=func(a)
      fb=func(b)
      if((fa.gt.0..and.fb.gt.0.).or.(fa.lt.0..and.fb.lt.0.))
     *     pause 'root must be bracketed for zbrent'
      c=b
      fc=fb
      do 11 iter=1,ITMAX
        if((fb.gt.0..and.fc.gt.0.).or.(fb.lt.0..and.fc.lt.0.))then
          c=a                          Rename a, b, c and adjust bounding interval d.
          fc=fa
          d=b-a
          e=d
        endif
        if(abs(fc).lt.abs(fb)) then
          a=b
          b=c
          c=a
          fa=fb
          fb=fc
          fc=fa
        endif
        tol1=2.*EPS*abs(b)+0.5*tol     Convergence check.
        xm=.5*(c-b)
        if(abs(xm).le.tol1 .or. fb.eq.0.)then
          zbrent=b
          return
        endif
        if(abs(e).ge.tol1 .and. abs(fa).gt.abs(fb)) then
          s=fb/fa                      Attempt inverse quadratic interpolation.
          if(a.eq.c) then
            p=2.*xm*s
            q=1.-s
          else
            q=fa/fc
            r=fb/fc
            p=s*(2.*xm*q*(q-r)-(b-a)*(r-1.))
            q=(q-1.)*(r-1.)*(s-1.)
```

```
        endif
        if(p.gt.0.) q=-q              Check whether in bounds.
        p=abs(p)
        if(2.*p .lt. min(3.*xm*q-abs(tol1*q),abs(e*q))) then
            e=d                       Accept interpolation.
            d=p/q
        else
            d=xm                      Interpolation failed, use bisection.
            e=d
        endif
    else                              Bounds decreasing too slowly, use bisection.
        d=xm
        e=d
    endif
    a=b                               Move last best guess to a.
    fa=fb
    if(abs(d) .gt. tol1) then         Evaluate new trial root.
        b=b+d
    else
        b=b+sign(tol1,xm)
    endif
    fb=func(b)
enddo 11
pause 'zbrent exceeding maximum iterations'
zbrent=b
return
END
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 3, 4. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §7.2.

9.4 Newton-Raphson Method Using Derivative

Perhaps the most celebrated of all one-dimensional root-finding routines is *Newton's method*, also called the *Newton-Raphson method*. This method is distinguished from the methods of previous sections by the fact that it requires the evaluation of both the function $f(x)$, *and* the derivative $f'(x)$, at arbitrary points x. The Newton-Raphson formula consists geometrically of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see Figure 9.4.1). Algebraically, the method derives from the familiar Taylor series expansion of a function in the neighborhood of a point,

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \dots . \tag{9.4.1}$$

For small enough values of δ, and for well-behaved functions, the terms beyond linear are unimportant, hence $f(x+\delta) = 0$ implies

$$\delta = -\frac{f(x)}{f'(x)}. \tag{9.4.2}$$

```
        endif
        if(p.gt.0.) q=-q              Check whether in bounds.
        p=abs(p)
        if(2.*p .lt. min(3.*xm*q-abs(tol1*q),abs(e*q))) then
            e=d                       Accept interpolation.
            d=p/q
        else
            d=xm                      Interpolation failed, use bisection.
            e=d
        endif
    else                              Bounds decreasing too slowly, use bisection.
        d=xm
        e=d
    endif
    a=b                               Move last best guess to a.
    fa=fb
    if(abs(d) .gt. tol1) then         Evaluate new trial root.
        b=b+d
    else
        b=b+sign(tol1,xm)
    endif
    fb=func(b)
enddo 11
pause 'zbrent exceeding maximum iterations'
zbrent=b
return
END
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 3, 4. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §7.2.

9.4 Newton-Raphson Method Using Derivative

Perhaps the most celebrated of all one-dimensional root-finding routines is *Newton's method*, also called the *Newton-Raphson method*. This method is distinguished from the methods of previous sections by the fact that it requires the evaluation of both the function $f(x)$, *and* the derivative $f'(x)$, at arbitrary points x. The Newton-Raphson formula consists geometrically of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see Figure 9.4.1). Algebraically, the method derives from the familiar Taylor series expansion of a function in the neighborhood of a point,

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \dots . \tag{9.4.1}$$

For small enough values of δ, and for well-behaved functions, the terms beyond linear are unimportant, hence $f(x+\delta) = 0$ implies

$$\delta = -\frac{f(x)}{f'(x)}. \tag{9.4.2}$$

Newton-Raphson is not restricted to one dimension. The method readily generalizes to multiple dimensions, as we shall see in §9.6 and §9.7, below.

Far from a root, where the higher-order terms in the series *are* important, the Newton-Raphson formula can give grossly inaccurate, meaningless corrections. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. This can be death to the method (see Figure 9.4.2). If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson sends its solution off to limbo, with vanishingly small hope of recovery. Like most powerful tools, Newton-Raphson can be destructive used in inappropriate circumstances. Figure 9.4.3 demonstrates another possible pathology.

Why do we call Newton-Raphson powerful? The answer lies in its rate of convergence: Within a small distance ϵ of x the function and its derivative are approximately:

$$\begin{aligned} f(x+\epsilon) &= f(x) + \epsilon f'(x) + \epsilon^2 \frac{f''(x)}{2} + \cdots, \\ f'(x+\epsilon) &= f'(x) + \epsilon f''(x) + \cdots \end{aligned} \tag{9.4.3}$$

By the Newton-Raphson formula,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{9.4.4}$$

so that

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)}. \tag{9.4.5}$$

When a trial solution x_i differs from the true root by ϵ_i, we can use (9.4.3) to express $f(x_i), f'(x_i)$ in (9.4.4) in terms of ϵ_i and derivatives at the root itself. The result is a recurrence relation for the deviations of the trial solutions

$$\epsilon_{i+1} = -\epsilon_i^2 \frac{f''(x)}{2f'(x)}. \tag{9.4.6}$$

Equation (9.4.6) says that Newton-Raphson converges *quadratically* (cf. equation 9.2.3). Near a root, the number of significant digits approximately *doubles* with each step. This very strong convergence property makes Newton-Raphson the method of choice for any function whose derivative can be evaluated efficiently, and whose derivative is continuous and nonzero in the neighborhood of a root.

Even where Newton-Raphson is rejected for the early stages of convergence (because of its poor global convergence properties), it is very common to "polish up" a root with one or two steps of Newton-Raphson, which can multiply by two or four its number of significant figures!

For an efficient realization of Newton-Raphson the user provides a routine that evaluates both $f(x)$ and its first derivative $f'(x)$ at the point x. The Newton-Raphson formula can also be applied using a numerical difference to approximate the true local derivative,

$$f'(x) \approx \frac{f(x+dx) - f(x)}{dx}. \tag{9.4.7}$$

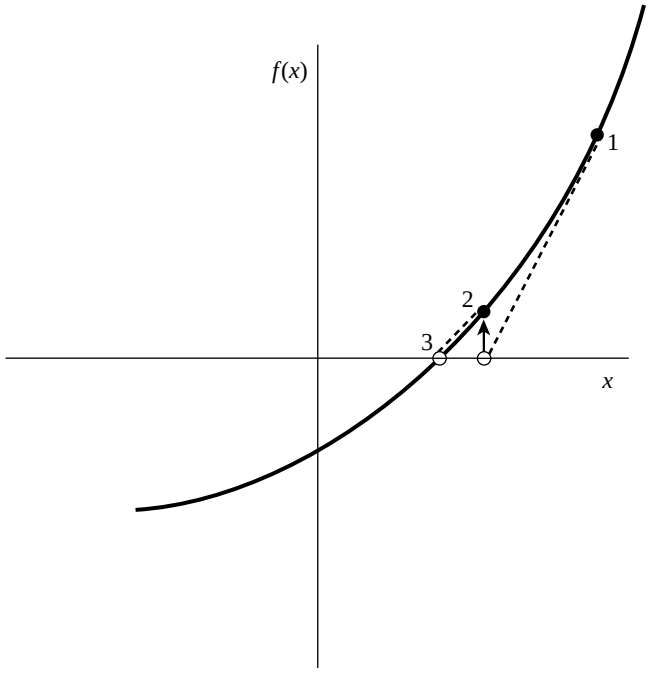


Figure 9.4.1. Newton's method extrapolates the local derivative to find the next estimate of the root. In this example it works well and converges quadratically.

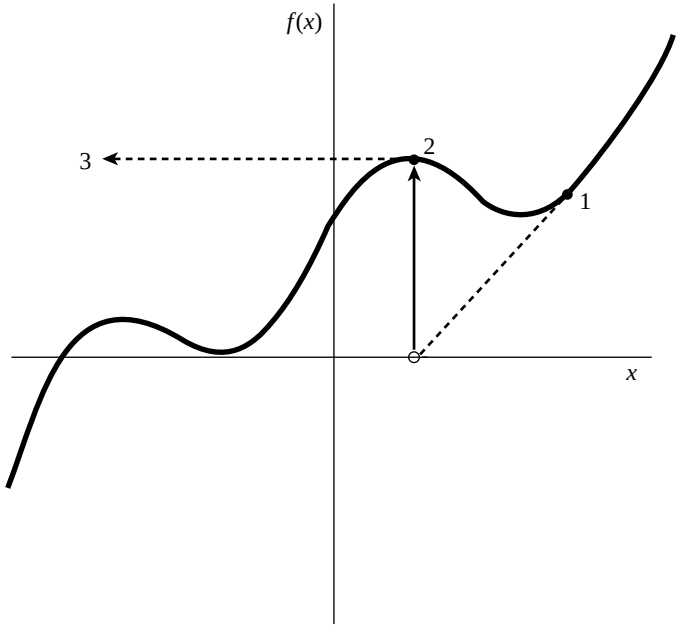


Figure 9.4.2. Unfortunate case where Newton's method encounters a local extremum and shoots off to outer space. Here bracketing bounds, as in `rtsafe`, would save the day.

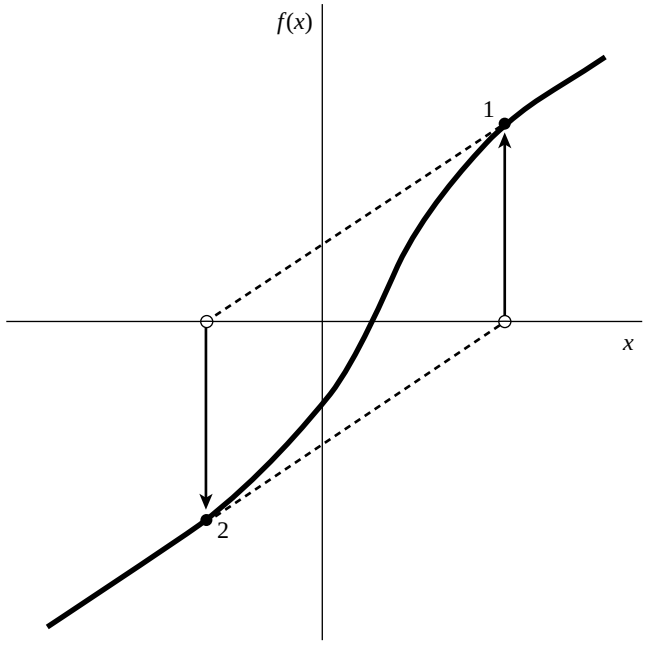


Figure 9.4.3. Unfortunate case where Newton's method enters a nonconvergent cycle. This behavior is often encountered when the function f is obtained, in whole or in part, by table interpolation. With a better initial guess, the method would have succeeded.

This is not, however, a recommended procedure for the following reasons: (i) You are doing two function evaluations per step, so *at best* the superlinear order of convergence will be only $\sqrt{2}$. (ii) If you take dx too small you will be wiped out by roundoff, while if you take it too large your order of convergence will be only linear, no better than using the *initial* evaluation $f'(x_0)$ for all subsequent steps. Therefore, Newton-Raphson with numerical derivatives is (in one dimension) always dominated by the secant method of §9.2. (In multidimensions, where there is a paucity of available methods, Newton-Raphson with numerical derivatives must be taken more seriously. See §§9.6–9.7.)

The following subroutine calls a user supplied subroutine `funcd(x,fn,df)` which returns the function value as `fn` and the derivative as `df`. We have included input bounds on the root simply to be consistent with previous root-finding routines: Newton does not adjust bounds, and works only on local information at the point `x`. The bounds are used only to pick the midpoint as the first guess, and to reject the solution if it wanders outside of the bounds.

```
FUNCTION rtnewt(funcd,x1,x2,xacc)
INTEGER JMAX
REAL rtnewt,x1,x2,xacc
EXTERNAL funcd
PARAMETER (JMAX=20)                    Set to maximum number of iterations.
    Using the Newton-Raphson method, find the root of a function known to lie in the interval
    [x1, x2]. The root rtnewt will be refined until its accuracy is known within ±xacc. funcd
    is a user-supplied subroutine that returns both the function value and the first derivative
    of the function at the point x.
INTEGER j
REAL df,dx,f
```

```
      rtnewt=.5*(x1+x2)                    Initial guess.
      do 11 j=1,JMAX
        call funcd(rtnewt,f,df)
        dx=f/df
        rtnewt=rtnewt-dx
        if((x1-rtnewt)*(rtnewt-x2).lt.0.)
     *       pause 'rtnewt jumped out of brackets'
        if(abs(dx).lt.xacc) return         Convergence.
      enddo 11
      pause 'rtnewt exceeded maximum iterations'
      END
```

While Newton-Raphson's global convergence properties are poor, it is fairly easy to design a fail-safe routine that utilizes a combination of bisection and Newton-Raphson. The hybrid algorithm takes a bisection step whenever Newton-Raphson would take the solution out of bounds, or whenever Newton-Raphson is not reducing the size of the brackets rapidly enough.

```
      FUNCTION rtsafe(funcd,x1,x2,xacc)
      INTEGER MAXIT
      REAL rtsafe,x1,x2,xacc
      EXTERNAL funcd
      PARAMETER (MAXIT=100)                Maximum allowed number of iterations.
        Using a combination of Newton-Raphson and bisection, find the root of a function bracketed
        between x1 and x2. The root, returned as the function value rtsafe, will be refined until
        its accuracy is known within ±xacc. funcd is a user-supplied subroutine which returns
        both the function value and the first derivative of the function.
      INTEGER j
      REAL df,dx,dxold,f,fh,fl,temp,xh,xl
      call funcd(x1,fl,df)
      call funcd(x2,fh,df)
      if((fl.gt.0..and.fh.gt.0.).or.(fl.lt.0..and.fh.lt.0.))
     *     pause 'root must be bracketed in rtsafe'
      if(fl.eq.0.)then
        rtsafe=x1
        return
      else if(fh.eq.0.)then
        rtsafe=x2
        return
      else if(fl.lt.0.)then                Orient the search so that f(xl) < 0.
        xl=x1
        xh=x2
      else
        xh=x1
        xl=x2
      endif
      rtsafe=.5*(x1+x2)                    Initialize the guess for root,
      dxold=abs(x2-x1)                     the "stepsize before last,"
      dx=dxold                             and the last step.
      call funcd(rtsafe,f,df)
      do 11 j=1,MAXIT                      Loop over allowed iterations.
        if(((rtsafe-xh)*df-f)*((rtsafe-xl)*df-f).gt.0.   Bisect if Newton out of range,
     *       .or. abs(2.*f).gt.abs(dxold*df) ) then      or not decreasing fast enough.
          dxold=dx
          dx=0.5*(xh-xl)
          rtsafe=xl+dx
          if(xl.eq.rtsafe)return           Change in root is negligible.
        else                               Newton step acceptable. Take it.
          dxold=dx
          dx=f/df
          temp=rtsafe
```

```
        rtsafe=rtsafe-dx
        if(temp.eq.rtsafe)return
    endif
    if(abs(dx).lt.xacc) return      Convergence criterion.
    call funcd(rtsafe,f,df)         The one new function evaluation per iteration.
    if(f.lt.0.) then                Maintain the bracket on the root.
        xl=rtsafe
    else
        xh=rtsafe
    endif
enddo 11
pause 'rtsafe exceeding maximum iterations'
return
END
```

For many functions the derivative $f'(x)$ often converges to machine accuracy before the function $f(x)$ itself does. When that is the case one need not subsequently update $f'(x)$. This shortcut is recommended only when you confidently understand the generic behavior of your function, but it speeds computations when the derivative calculation is laborious. (Formally this makes the convergence only linear, but if the derivative isn't changing anyway, you can do no better.)

Newton-Raphson and Fractals

An interesting sidelight to our repeated warnings about Newton-Raphson's unpredictable global convergence properties — its very rapid local convergence notwithstanding — is to investigate, for some particular equation, the set of starting values from which the method does, or doesn't converge to a root.

Consider the simple equation

$$z^3 - 1 = 0 \tag{9.4.8}$$

whose single real root is $z = 1$, but which also has complex roots at the other two cube roots of unity, $\exp(\pm 2\pi i/3)$. Newton's method gives the iteration

$$z_{j+1} = z_j - \frac{z_j^3 - 1}{3z_j^2} \tag{9.4.9}$$

Up to now, we have applied an iteration like equation (9.4.9) only for real starting values z_0, but in fact all of the equations in this section also apply in the complex plane. We can therefore map out the complex plane into regions from which a starting value z_0, iterated in equation (9.4.9), will, or won't, converge to $z = 1$. Naively, we might expect to find a "basin of convergence" somehow surrounding the root $z = 1$. We surely do not expect the basin of convergence to fill the whole plane, because the plane must also contain regions that converge to each of the two complex roots. In fact, by symmetry, the three regions must have identical shapes. Perhaps they will be three symmetric 120° wedges, with one root centered in each?

Now take a look at Figure 9.4.4, which shows the result of a numerical exploration. The basin of convergence does indeed cover $1/3$ the area of the complex plane, but its boundary is highly irregular — in fact, *fractal*. (A fractal, so called, has self-similar structure that repeats on all scales of magnification.) How

Figure 9.4.4. The complex z plane with real and imaginary components in the range $(-2, 2)$. The black region is the set of points from which Newton's method converges to the root $z = 1$ of the equation $z^3 - 1 = 0$. Its shape is fractal.

does this fractal emerge from something as simple as Newton's method, and an equation as simple as (9.4.8)? The answer is already implicit in Figure 9.4.2, which showed how, on the real line, a local extremum causes Newton's method to shoot off to infinity. Suppose one is *slightly* removed from such a point. Then one might be shot off not to infinity, but — by luck — right into the basin of convergence of the desired root. But that means that in the neighborhood of an extremum there must be a tiny, perhaps distorted, copy of the basin of convergence — a kind of "one-bounce away" copy. Similar logic shows that there can be "two-bounce" copies, "three-bounce" copies, and so on. A fractal thus emerges.

Notice that, for equation (9.4.8), almost the whole real axis is in the domain of convergence for the root $z = 1$. We say "almost" because of the peculiar discrete points on the negative real axis whose convergence is indeterminate (see figure). What happens if you start Newton's method from one of these points? (Try it.)

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.4.

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).

Mandelbrot, B.B. 1983, *The Fractal Geometry of Nature* (San Francisco: W.H. Freeman).

Peitgen, H.-O., and Saupe, D. (eds.) 1988, *The Science of Fractal Images* (New York: Springer-Verlag).

9.5 Roots of Polynomials

Here we present a few methods for finding roots of polynomials. These will serve for most practical problems involving polynomials of low-to-moderate degree or for well-conditioned polynomials of higher degree. Not as well appreciated as it ought to be is the fact that some polynomials are exceedingly ill-conditioned. The tiniest changes in a polynomial's coefficients can, in the worst case, send its roots sprawling all over the complex plane. (An infamous example due to Wilkinson is detailed by Acton [1].)

Recall that a polynomial of degree n will have n roots. The roots can be real or complex, and they might not be distinct. If the coefficients of the polynomial are real, then complex roots will occur in pairs that are conjugate, i.e., if $x_1 = a + bi$ is a root then $x_2 = a - bi$ will also be a root. When the coefficients are complex, the complex roots need not be related.

Multiple roots, or closely spaced roots, produce the most difficulty for numerical algorithms (see Figure 9.5.1). For example, $P(x) = (x - a)^2$ has a double real root at $x = a$. However, we cannot bracket the root by the usual technique of identifying neighborhoods where the function changes sign, nor will slope-following methods such as Newton-Raphson work well, because both the function and its derivative vanish at a multiple root. Newton-Raphson *may* work, but slowly, since large roundoff errors can occur. When a root is known in advance to be multiple, then special methods of attack are readily devised. Problems arise when (as is generally the case) we do not know in advance what pathology a root will display.

Deflation of Polynomials

When seeking several or all roots of a polynomial, the total effort can be significantly reduced by the use of *deflation*. As each root r is found, the polynomial is factored into a product involving the root and a reduced polynomial of degree one less than the original, i.e., $P(x) = (x - r)Q(x)$. Since the roots of Q are exactly the remaining roots of P, the effort of finding additional roots decreases, because we work with polynomials of lower and lower degree as we find successive roots. Even more important, with deflation we can avoid the blunder of having our iterative method converge twice to the same (nonmultiple) root instead of separately to two different roots.

Deflation, which amounts to synthetic division, is a simple operation that acts on the array of polynomial coefficients. The concise code for synthetic division by a monomial factor was given in §5.3 above. You can deflate complex roots either by converting that code to complex data type, or else — in the case of a polynomial with real coefficients but possibly complex roots — by deflating by a quadratic factor,

$$[x - (a + ib)]\,[x - (a - ib)] = x^2 - 2ax + (a^2 + b^2) \tag{9.5.1}$$

Peitgen, H.-O., and Saupe, D. (eds.) 1988, *The Science of Fractal Images* (New York: Springer-Verlag).

9.5 Roots of Polynomials

Here we present a few methods for finding roots of polynomials. These will serve for most practical problems involving polynomials of low-to-moderate degree or for well-conditioned polynomials of higher degree. Not as well appreciated as it ought to be is the fact that some polynomials are exceedingly ill-conditioned. The tiniest changes in a polynomial's coefficients can, in the worst case, send its roots sprawling all over the complex plane. (An infamous example due to Wilkinson is detailed by Acton [1].)

Recall that a polynomial of degree n will have n roots. The roots can be real or complex, and they might not be distinct. If the coefficients of the polynomial are real, then complex roots will occur in pairs that are conjugate, i.e., if $x_1 = a + bi$ is a root then $x_2 = a - bi$ will also be a root. When the coefficients are complex, the complex roots need not be related.

Multiple roots, or closely spaced roots, produce the most difficulty for numerical algorithms (see Figure 9.5.1). For example, $P(x) = (x - a)^2$ has a double real root at $x = a$. However, we cannot bracket the root by the usual technique of identifying neighborhoods where the function changes sign, nor will slope-following methods such as Newton-Raphson work well, because both the function and its derivative vanish at a multiple root. Newton-Raphson *may* work, but slowly, since large roundoff errors can occur. When a root is known in advance to be multiple, then special methods of attack are readily devised. Problems arise when (as is generally the case) we do not know in advance what pathology a root will display.

Deflation of Polynomials

When seeking several or all roots of a polynomial, the total effort can be significantly reduced by the use of *deflation*. As each root r is found, the polynomial is factored into a product involving the root and a reduced polynomial of degree one less than the original, i.e., $P(x) = (x - r)Q(x)$. Since the roots of Q are exactly the remaining roots of P, the effort of finding additional roots decreases, because we work with polynomials of lower and lower degree as we find successive roots. Even more important, with deflation we can avoid the blunder of having our iterative method converge twice to the same (nonmultiple) root instead of separately to two different roots.

Deflation, which amounts to synthetic division, is a simple operation that acts on the array of polynomial coefficients. The concise code for synthetic division by a monomial factor was given in §5.3 above. You can deflate complex roots either by converting that code to complex data type, or else — in the case of a polynomial with real coefficients but possibly complex roots — by deflating by a quadratic factor,

$$[x - (a + ib)]\,[x - (a - ib)] = x^2 - 2ax + (a^2 + b^2) \tag{9.5.1}$$

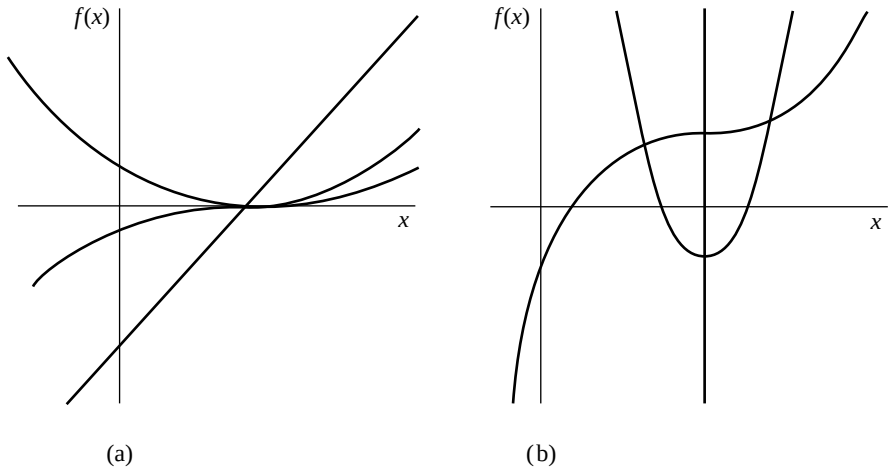


Figure 9.5.1. (a) Linear, quadratic, and cubic behavior at the roots of polynomials. Only under high magnification (b) does it become apparent that the cubic has one, not three, roots, and that the quadratic has two roots rather than none.

The routine `poldiv` in §5.3 can be used to divide the polynomial by this factor.

Deflation must, however, be utilized with care. Because each new root is known with only finite accuracy, errors creep into the determination of the coefficients of the successively deflated polynomial. Consequently, the roots can become more and more inaccurate. It matters a lot whether the inaccuracy creeps in stably (plus or minus a few multiples of the machine precision at each stage) or unstably (erosion of successive significant figures until the results become meaningless). Which behavior occurs depends on just how the root is divided out. *Forward deflation*, where the new polynomial coefficients are computed in the order from the highest power of x down to the constant term, was illustrated in §5.3. This turns out to be stable if the root of smallest absolute value is divided out at each stage. Alternatively, one can do *backward deflation*, where new coefficients are computed in order from the constant term up to the coefficient of the highest power of x. This is stable if the remaining root of *largest* absolute value is divided out at each stage.

A polynomial whose coefficients are interchanged "end-to-end," so that the constant becomes the highest coefficient, etc., has its roots mapped into their reciprocals. (Proof: Divide the whole polynomial by its highest power x^n and rewrite it as a polynomial in $1/x$.) The algorithm for backward deflation is therefore virtually identical to that of forward deflation, except that the original coefficients are taken in reverse order and the reciprocal of the deflating root is used. Since we will use forward deflation below, we leave to you the exercise of writing a concise coding for backward deflation (as in §5.3). For more on the stability of deflation, consult [2].

To minimize the impact of increasing errors (even stable ones) when using deflation, it is advisable to treat roots of the successively deflated polynomials as only *tentative* roots of the original polynomial. One then *polishes* these tentative roots by taking them as initial guesses that are to be re-solved for, using the *nondeflated* original polynomial P. Again you must beware lest two deflated roots are inaccurate enough that, under polishing, they both converge to the same undeflated root; in that case you gain a spurious root-multiplicity and lose a distinct root. This is detectable,

since you can compare each polished root for equality to previous ones from distinct tentative roots. When it happens, you are advised to deflate the polynomial just once (and for this root only), then again polish the tentative root, or to use Maehly's procedure (see equation 9.5.29 below).

Below we say more about techniques for polishing real and complex-conjugate tentative roots. First, let's get back to overall strategy.

There are two schools of thought about how to proceed when faced with a polynomial of real coefficients. One school says to go after the easiest quarry, the real, distinct roots, by the same kinds of methods that we have discussed in previous sections for general functions, i.e., trial-and-error bracketing followed by a safe Newton-Raphson as in `rtsafe`. Sometimes you are *only* interested in real roots, in which case the strategy is complete. Otherwise, you then go after quadratic factors of the form (9.5.1) by any of a variety of methods. One such is Bairstow's method, which we will discuss below in the context of root polishing. Another is Muller's method, which we here briefly discuss.

Muller's Method

Muller's method generalizes the secant method, but uses quadratic interpolation among three points instead of linear interpolation between two. Solving for the zeros of the quadratic allows the method to find complex pairs of roots. Given *three* previous guesses for the root x_{i-2}, x_{i-1}, x_i, and the values of the polynomial $P(x)$ at those points, the next approximation x_{i+1} is produced by the following formulas,

$$\begin{aligned} q &\equiv \frac{x_i - x_{i-1}}{x_{i-1} - x_{i-2}} \\ A &\equiv qP(x_i) - q(1+q)P(x_{i-1}) + q^2 P(x_{i-2}) \\ B &\equiv (2q+1)P(x_i) - (1+q)^2 P(x_{i-1}) + q^2 P(x_{i-2}) \\ C &\equiv (1+q)P(x_i) \end{aligned} \tag{9.5.2}$$

followed by

$$x_{i+1} = x_i - (x_i - x_{i-1}) \left[\frac{2C}{B \pm \sqrt{B^2 - 4AC}} \right] \tag{9.5.3}$$

where the sign in the denominator is chosen to make its absolute value or modulus as large as possible. You can start the iterations with any three values of x that you like, e.g., three equally spaced values on the real axis. Note that you must allow for the possibility of a complex denominator, and subsequent complex arithmetic, in implementing the method.

Muller's method is sometimes also used for finding complex zeros of analytic functions (not just polynomials) in the complex plane, for example in the IMSL routine `ZANLY` [3].

Laguerre's Method

The second school regarding overall strategy happens to be the one to which we belong. That school advises you to use one of a very small number of methods that will converge (though with greater or lesser efficiency) to all types of roots: real, complex, single, or multiple. Use such a method to get tentative values for all n roots of your nth degree polynomial. Then go back and polish them as you desire.

Laguerre's method is by far the most straightforward of these general, complex methods. It does require complex arithmetic, even while converging to real roots; however, for polynomials with all real roots, it is guaranteed to converge to a root from any starting point. For polynomials with some complex roots, little is theoretically proved about the method's convergence. Much empirical experience, however, suggests that nonconvergence is extremely unusual, and, further, can almost always be fixed by a simple scheme to break a nonconverging limit cycle. (This is implemented in our routine, below.) An example of a polynomial that requires this cycle-breaking scheme is one of high degree ($\gtrsim 20$), with all its roots just outside of the complex unit circle, approximately equally spaced around it. When the method converges on a simple complex zero, it is known that its convergence is third order.

In some instances the complex arithmetic in the Laguerre method is no disadvantage, since the polynomial itself may have complex coefficients.

To motivate (although not rigorously derive) the Laguerre formulas we can note the following relations between the polynomial and its roots and derivatives

$$P_n(x) = (x - x_1)(x - x_2)\ldots(x - x_n) \tag{9.5.4}$$

$$\ln|P_n(x)| = \ln|x - x_1| + \ln|x - x_2| + \ldots + \ln|x - x_n| \tag{9.5.5}$$

$$\frac{d\ln|P_n(x)|}{dx} = +\frac{1}{x - x_1} + \frac{1}{x - x_2} + \ldots + \frac{1}{x - x_n} = \frac{P_n'}{P_n} \equiv G \tag{9.5.6}$$

$$\begin{aligned} -\frac{d^2\ln|P_n(x)|}{dx^2} &= +\frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \ldots + \frac{1}{(x - x_n)^2} \\ &= \left[\frac{P_n'}{P_n}\right]^2 - \frac{P_n''}{P_n} \equiv H \end{aligned} \tag{9.5.7}$$

Starting from these relations, the Laguerre formulas make what Acton [1] nicely calls "a rather drastic set of assumptions": The root x_1 that we seek is assumed to be located some distance a from our current guess x, while *all other roots* are assumed to be located at a distance b

$$x - x_1 = a \quad ; \quad x - x_i = b \quad i = 2, 3, \ldots, n \tag{9.5.8}$$

Then we can express (9.5.6), (9.5.7) as

$$\frac{1}{a} + \frac{n-1}{b} = G \tag{9.5.9}$$

$$\frac{1}{a^2} + \frac{n-1}{b^2} = H \tag{9.5.10}$$

which yields as the solution for a

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}} \tag{9.5.11}$$

where the sign should be taken to yield the largest magnitude for the denominator. Since the factor inside the square root can be negative, a can be complex. (A more rigorous justification of equation 9.5.11 is in [4].)

The method operates iteratively: For a trial value x, a is calculated by equation (9.5.11). Then $x - a$ becomes the next trial value. This continues until a is sufficiently small.

The following routine implements the Laguerre method to find one root of a given polynomial of degree m, whose coefficients can be complex. As usual, the first coefficient a(1) is the constant term, while a(m+1) is the coefficient of the highest power of x. The routine implements a simplified version of an elegant stopping criterion due to Adams [5], which neatly balances the desire to achieve full machine accuracy, on the one hand, with the danger of iterating forever in the presence of roundoff error, on the other.

```
SUBROUTINE laguer(a,m,x,its)
INTEGER m,its,MAXIT,MR,MT
REAL EPSS
COMPLEX a(m+1),x
PARAMETER (EPSS=2.e-7,MR=8,MT=10,MAXIT=MT*MR)
    Given the degree m and the complex coefficients a(1:m+1) of the polynomial sum_{i=1}^{m+1} a(i)x^{i-1},
    and given a complex value x, this routine improves x by Laguerre's method until it converges, within the achievable roundoff limit, to a root of the given polynomial. The number of iterations taken is returned as its.
    Parameters: EPSS is the estimated fractional roundoff error. We try to break (rare) limit cycles with MR different fractional values, once every MT steps, for MAXIT total allowed iterations.
INTEGER iter,j
REAL abx,abp,abm,err,frac(MR)
COMPLEX dx,x1,b,d,f,g,h,sq,gp,gm,g2
SAVE frac
DATA frac /.5,.25,.75,.13,.38,.62,.88,1./      Fractions used to break a limit cycle.
do 12 iter=1,MAXIT                             Loop over iterations up to allowed maximum.
    its=iter
    b=a(m+1)
    err=abs(b)
    d=cmplx(0.,0.)
    f=cmplx(0.,0.)
    abx=abs(x)
    do 11 j=m,1,-1                             Efficient computation of the polynomial and its first
        f=x*f+d                                    two derivatives.
        d=x*d+b
        b=x*b+a(j)
        err=abs(b)+abx*err
    enddo 11
    err=EPSS*err                               Estimate of roundoff error in evaluating polynomial.
    if(abs(b).le.err) then                     We are on the root.
        return
    else                                       The generic case: use Laguerre's formula.
        g=d/b
        g2=g*g
        h=g2-2.*f/b
        sq=sqrt((m-1)*(m*h-g2))
        gp=g+sq
        gm=g-sq
        abp=abs(gp)
        abm=abs(gm)
        if(abp.lt.abm) gp=gm
        if (max(abp,abm).gt.0.) then
            dx=m/gp
```

```
        else
            dx=exp(cmplx(log(1.+abx),float(iter)))
        endif
    endif
    x1=x-dx
    if(x.eq.x1)return                Converged.
    if (mod(iter,MT).ne.0) then
        x=x1
    else                             Every so often we take a fractional step, to break any
        x=x-dx*frac(iter/MT)             limit cycle (itself a rare occurrence).
    endif
enddo 12
pause 'too many iterations in laguer'  Very unusual — can occur only for complex roots.
return                               Try a different starting guess for the root.
END
```

Here is a driver routine that calls `laguer` in succession for each root, performs the deflation, optionally polishes the roots by the same Laguerre method — if you are not going to polish in some other way — and finally sorts the roots by their real parts. (We will use this routine in Chapter 13.)

```
SUBROUTINE zroots(a,m,roots,polish)
INTEGER m,MAXM
REAL EPS
COMPLEX a(m+1),roots(m)
LOGICAL polish
PARAMETER (EPS=1.e-6,MAXM=101)  A small number and maximum anticipated value of m+1.
C  USES laguer
    Given the degree m and the complex coefficients a(1:m+1) of the polynomial sum_{i=1}^{m+1} a(i)x^{i-1},
    this routine successively calls laguer and finds all m complex roots. The logical variable
    polish should be input as .true. if polishing (also by Laguerre's method) is desired,
    .false. if the roots will be subsequently polished by other means.
INTEGER i,j,jj,its
COMPLEX ad(MAXM),x,b,c
do 11 j=1,m+1                    Copy of coefficients for successive deflation.
    ad(j)=a(j)
enddo 11
do 13 j=m,1,-1                   Loop over each root to be found.
    x=cmplx(0.,0.)               Start at zero to favor convergence to smallest remaining root.
    call laguer(ad,j,x,its)      Find the root.
    if(abs(aimag(x)).le.2.*EPS**2*abs(real(x))) x=cmplx(real(x),0.)
    roots(j)=x
    b=ad(j+1)                    Forward deflation.
    do 12 jj=j,1,-1
        c=ad(jj)
        ad(jj)=b
        b=x*b+c
    enddo 12
enddo 13
if (polish) then
    do 14 j=1,m                  Polish the roots using the undeflated coefficients.
        call laguer(a,m,roots(j),its)
    enddo 14
endif
do 16 j=2,m                      Sort roots by their real parts by straight insertion.
    x=roots(j)
    do 15 i=j-1,1,-1
        if(real(roots(i)).le.real(x))goto 10
        roots(i+1)=roots(i)
    enddo 15
    i=0
```

```
10      roots(i+1)=x
    enddo 16
    return
    END
```

Eigenvalue Methods

The eigenvalues of a matrix **A** are the roots of the "characteristic polynomial" $P(x) = \det[\mathbf{A} - x\mathbf{I}]$. However, as we will see in Chapter 11, root-finding is not generally an efficient way to find eigenvalues. Turning matters around, we can use the more efficient eigenvalue methods that are discussed in Chapter 11 to find the roots of arbitrary polynomials. You can easily verify (see, e.g., [6]) that the characteristic polynomial of the special $m \times m$ *companion matrix*

$$\mathbf{A} = \begin{pmatrix} -\frac{a_m}{a_{m+1}} & -\frac{a_{m-1}}{a_{m+1}} & \cdots & -\frac{a_2}{a_{m+1}} & -\frac{a_1}{a_{m+1}} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \tag{9.5.12}$$

is equivalent to the general polynomial

$$P(x) = \sum_{i=1}^{m+1} a_i x^{i-1} \tag{9.5.13}$$

If the coefficients a_i are real, rather than complex, then the eigenvalues of **A** can be found using the routines `balanc` and `hqr` in §§11.5–11.6 (see discussion there). This method, implemented in the routine `zrhqr` following, is typically about a factor 2 slower than `zroots` (above). However, for some classes of polynomials, it is a more robust technique, largely because of the fairly sophisticated convergence methods embodied in `hqr`. If your polynomial has real coefficients, and you are having trouble with `zroots`, then `zrhqr` is a recommended alternative.

```
    SUBROUTINE zrhqr(a,m,rtr,rti)
    INTEGER m,MAXM
    REAL a(m+1),rtr(m),rti(m)
    PARAMETER (MAXM=50)
C   USES balanc,hqr
```

Find all the roots of a polynomial with real coefficients, $\sum_{i=1}^{\mathtt{m+1}} \mathtt{a}(i)x^{i-1}$, given the degree `m` and the coefficients `a(1:m+1)`. The method is to construct an upper Hessenberg matrix whose eigenvalues are the desired roots, and then use the routines `balanc` and `hqr`. The real and imaginary parts of the roots are returned in `rtr(1:m)` and `rti(1:m)`, respectively.

```
    INTEGER j,k
    REAL hess(MAXM,MAXM),xr,xi
    if (m.gt.MAXM.or.a(m+1).eq.0.) pause 'bad args in zrhqr'
    do 12 k=1,m                    Construct the matrix.
        hess(1,k)=-a(m+1-k)/a(m+1)
        do 11 j=2,m
            hess(j,k)=0.
        enddo 11
        if (k.ne.m) hess(k+1,k)=1.
```

```
      enddo 12
      call balanc(hess,m,MAXM)     Find its eigenvalues.
      call hqr(hess,m,MAXM,rtr,rti)
      do 14 j=2,m                  Sort roots by their real parts by straight insertion.
          xr=rtr(j)
          xi=rti(j)
          do 13 k=j-1,1,-1
              if(rtr(k).le.xr)goto 1
              rtr(k+1)=rtr(k)
              rti(k+1)=rti(k)
          enddo 13
          k=0
1         rtr(k+1)=xr
          rti(k+1)=xi
      enddo 14
      return
      END
```

Other Sure-Fire Techniques

The *Jenkins-Traub method* has become practically a standard in black-box polynomial root-finders, e.g., in the IMSL library [3]. The method is too complicated to discuss here, but is detailed, with references to the primary literature, in [4].

The *Lehmer-Schur algorithm* is one of a class of methods that isolate roots in the complex plane by generalizing the notion of one-dimensional bracketing. It is possible to determine efficiently whether there are any polynomial roots within a circle of given center and radius. From then on it is a matter of bookkeeping to hunt down all the roots by a series of decisions regarding where to place new trial circles. Consult [1] for an introduction.

Techniques for Root-Polishing

Newton-Raphson works very well for real roots once the neighborhood of a root has been identified. The polynomial and its derivative can be efficiently simultaneously evaluated as in §5.3. For a polynomial of degree `n-1` with coefficients `c(1)...c(n)`, the following segment of code embodies one cycle of Newton-Raphson:

```
      p=c(n)*x+c(n-1)
      p1=c(n)
      do 11 i=n-2,1,-1
          p1=p+p1*x
          p=c(i)+p*x
      enddo 11
      if (p1.eq.0.) pause 'derivative should not vanish'
      x=x-p/p1
```

Once all real roots of a polynomial have been polished, one must polish the complex roots, either directly, or by looking for quadratic factors.

Direct polishing by Newton-Raphson is straightforward for complex roots if the above code is converted to complex data types. With real polynomial coefficients, note that your starting guess (tentative root) *must* be off the real axis, otherwise you will never get off that axis — and may get shot off to infinity by a minimum or maximum of the polynomial.

For real polynomials, the alternative means of polishing complex roots (or, for that matter, double real roots) is *Bairstow's method*, which seeks quadratic factors. The advantage of going after quadratic factors is that it avoids all complex arithmetic. Bairstow's method seeks a quadratic factor that embodies the two roots $x = a \pm ib$, namely

$$x^2 - 2ax + (a^2 + b^2) \equiv x^2 + Bx + C \tag{9.5.14}$$

In general if we divide a polynomial by a quadratic factor, there will be a linear remainder

$$P(x) = (x^2 + Bx + C)Q(x) + Rx + S. \tag{9.5.15}$$

Given B and C, R and S can be readily found, by polynomial division (§5.3). We can consider R and S to be adjustable functions of B and C, and they will be zero if the quadratic factor is zero.

In the neighborhood of a root a first-order Taylor series expansion approximates the variation of R, S with respect to small changes in B, C

$$R(B + \delta B, C + \delta C) \approx R(B, C) + \frac{\partial R}{\partial B}\delta B + \frac{\partial R}{\partial C}\delta C \tag{9.5.16}$$

$$S(B + \delta B, C + \delta C) \approx S(B, C) + \frac{\partial S}{\partial B}\delta B + \frac{\partial S}{\partial C}\delta C \tag{9.5.17}$$

To evaluate the partial derivatives, consider the derivative of (9.5.15) with respect to C. Since $P(x)$ is a fixed polynomial, it is independent of C, hence

$$0 = (x^2 + Bx + C)\frac{\partial Q}{\partial C} + Q(x) + \frac{\partial R}{\partial C}x + \frac{\partial S}{\partial C} \tag{9.5.18}$$

which can be rewritten as

$$-Q(x) = (x^2 + Bx + C)\frac{\partial Q}{\partial C} + \frac{\partial R}{\partial C}x + \frac{\partial S}{\partial C} \tag{9.5.19}$$

Similarly, $P(x)$ is independent of B, so differentiating (9.5.15) with respect to B gives

$$-xQ(x) = (x^2 + Bx + C)\frac{\partial Q}{\partial B} + \frac{\partial R}{\partial B}x + \frac{\partial S}{\partial B} \tag{9.5.20}$$

Now note that equation (9.5.19) matches equation (9.5.15) in form. Thus if we perform a second synthetic division of $P(x)$, i.e., a division of $Q(x)$, yielding a remainder $R_1 x + S_1$, then

$$\frac{\partial R}{\partial C} = -R_1 \qquad \frac{\partial S}{\partial C} = -S_1 \tag{9.5.21}$$

To get the remaining partial derivatives, evaluate equation (9.5.20) at the two roots of the quadratic, x_+ and x_-. Since

$$Q(x_\pm) = R_1 x_\pm + S_1 \tag{9.5.22}$$

we get

$$\frac{\partial R}{\partial B}x_+ + \frac{\partial S}{\partial B} = -x_+(R_1 x_+ + S_1) \tag{9.5.23}$$

$$\frac{\partial R}{\partial B}x_- + \frac{\partial S}{\partial B} = -x_-(R_1 x_- + S_1) \tag{9.5.24}$$

Solve these two equations for the partial derivatives, using

$$x_+ + x_- = -B \qquad x_+ x_- = C \tag{9.5.25}$$

and find

$$\frac{\partial R}{\partial B} = BR_1 - S_1 \qquad \frac{\partial S}{\partial B} = CR_1 \tag{9.5.26}$$

Bairstow's method now consists of using Newton-Raphson in two dimensions (which is actually the subject of the *next* section) to find a simultaneous zero of R and S. Synthetic division is used twice per cycle to evaluate R, S and their partial derivatives with respect to B, C. Like one-dimensional Newton-Raphson, the method works well in the vicinity of a root pair (real or complex), but it can fail miserably when started at a random point. We therefore recommend it only in the context of polishing tentative complex roots.

```
      SUBROUTINE qroot(p,n,b,c,eps)
      INTEGER n,NMAX,ITMAX
      REAL b,c,eps,p(n),TINY
      PARAMETER (NMAX=20,ITMAX=20,TINY=1.0e-6)
C     USES poldiv
        Given coefficients p(1:n) of a polynomial of degree n-1, and trial values for the coefficients
        of a quadratic factor x*x+b*x+c, improve the solution until the coefficients b,c change
        by less than eps. The routine poldiv §5.3 is used.
        Parameters: At most NMAX coefficients, ITMAX iterations.
      INTEGER iter
      REAL delb,delc,div,r,rb,rc,s,sb,sc,d(3),q(NMAX),qq(NMAX),rem(NMAX)
      d(3)=1.
      do 11 iter=1,ITMAX
          d(2)=b
          d(1)=c
          call poldiv(p,n,d,3,q,rem)
          s=rem(1)                              First division r,s.
          r=rem(2)
          call poldiv(q,n-1,d,3,qq,rem)
          sc=-rem(1)                            Second division partial r,s with respect to c.
          rc=-rem(2)
          sb=-c*rc
          rb=sc-b*rc
          div=1./(sb*rc-sc*rb)                  Solve 2x2 equation.
          delb=(r*sc-s*rc)*div
          delc=(-r*sb+s*rb)*div
          b=b+delb
          c=c+delc
          if((abs(delb).le.eps*abs(b).or.abs(b).lt.TINY)
     *         .and.(abs(delc).le.eps*abs(c)
     *         .or.abs(c).lt.TINY)) return    Coefficients converged.
      enddo 11
      pause 'too many iterations in qroot'
      END
```

We have already remarked on the annoyance of having two tentative roots collapse to one value under polishing. You are left not knowing whether your polishing procedure has lost a root, or whether there *is* actually a double root, which was split only by roundoff errors in your previous deflation. One solution is deflate-and-repolish; but deflation is what we are trying to avoid at the polishing stage. An alternative is *Maehly's procedure*. Maehly pointed out that the derivative of the reduced polynomial

$$P_j(x) \equiv \frac{P(x)}{(x-x_1)\cdots(x-x_j)} \tag{9.5.27}$$

can be written as

$$P_j'(x) = \frac{P'(x)}{(x-x_1)\cdots(x-x_j)} - \frac{P(x)}{(x-x_1)\cdots(x-x_j)} \sum_{i=1}^{j} (x-x_i)^{-1} \tag{9.5.28}$$

Hence one step of Newton-Raphson, taking a guess x_k into a new guess x_{k+1}, can be written as

$$x_{k+1} = x_k - \frac{P(x_k)}{P'(x_k) - P(x_k)\sum_{i=1}^{j}(x_k - x_i)^{-1}} \tag{9.5.29}$$

This equation, if used with i ranging over the roots already polished, will prevent a tentative root from spuriously hopping to another one's true root. It is an example of so-called *zero suppression* as an alternative to true deflation.

Muller's method, which was described above, can also be useful at the polishing stage.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 7. [1]

Peters G., and Wilkinson, J.H. 1971, *Journal of the Institute of Mathematics and its Applications*, vol. 8, pp. 16–35. [2]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [3]

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.9–8.13. [4]

Adams, D.A. 1967, *Communications of the ACM*, vol. 10, pp. 655–658. [5]

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §4.4.3. [6]

Henrici, P. 1974, *Applied and Computational Complex Analysis*, vol. 1 (New York: Wiley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§5.5–5.9.

9.6 Newton-Raphson Method for Nonlinear Systems of Equations

We make an extreme, but wholly defensible, statement: There are *no* good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely) there *never will be* any good, general methods: Consider the case of two dimensions, where we want to solve simultaneously

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \end{aligned} \tag{9.6.1}$$

The functions f and g are two arbitrary functions, each of which has zero contour lines that divide the (x,y) plane into regions where their respective function is positive or negative. These zero contour boundaries are of interest to us. The solutions that we seek are those points (if any) that are common to the zero contours of f and g (see Figure 9.6.1). Unfortunately, the functions f and g have, in general, no relation to each other at all! There is nothing special about a common point from either f's point of view, or from g's. In order to find all common points, which are the solutions of our nonlinear equations, we will (in general) have to do neither more nor less than map out the full zero contours of both functions. Note further that the zero contours will (in general) consist of an unknown number of disjoint closed curves. How can we ever hope to know when we have found all such disjoint pieces?

For problems in more than two dimensions, we need to find points mutually common to N unrelated zero-contour hypersurfaces, each of dimension $N-1$.

This equation, if used with i ranging over the roots already polished, will prevent a tentative root from spuriously hopping to another one's true root. It is an example of so-called *zero suppression* as an alternative to true deflation.

Muller's method, which was described above, can also be useful at the polishing stage.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 7. [1]

Peters G., and Wilkinson, J.H. 1971, *Journal of the Institute of Mathematics and its Applications*, vol. 8, pp. 16–35. [2]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [3]

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.9–8.13. [4]

Adams, D.A. 1967, *Communications of the ACM*, vol. 10, pp. 655–658. [5]

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §4.4.3. [6]

Henrici, P. 1974, *Applied and Computational Complex Analysis*, vol. 1 (New York: Wiley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§5.5–5.9.

9.6 Newton-Raphson Method for Nonlinear Systems of Equations

We make an extreme, but wholly defensible, statement: There are *no* good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely) there *never will be* any good, general methods: Consider the case of two dimensions, where we want to solve simultaneously

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \end{aligned} \tag{9.6.1}$$

The functions f and g are two arbitrary functions, each of which has zero contour lines that divide the (x,y) plane into regions where their respective function is positive or negative. These zero contour boundaries are of interest to us. The solutions that we seek are those points (if any) that are common to the zero contours of f and g (see Figure 9.6.1). Unfortunately, the functions f and g have, in general, no relation to each other at all! There is nothing special about a common point from either f's point of view, or from g's. In order to find all common points, which are the solutions of our nonlinear equations, we will (in general) have to do neither more nor less than map out the full zero contours of both functions. Note further that the zero contours will (in general) consist of an unknown number of disjoint closed curves. How can we ever hope to know when we have found all such disjoint pieces?

For problems in more than two dimensions, we need to find points mutually common to N unrelated zero-contour hypersurfaces, each of dimension $N-1$.

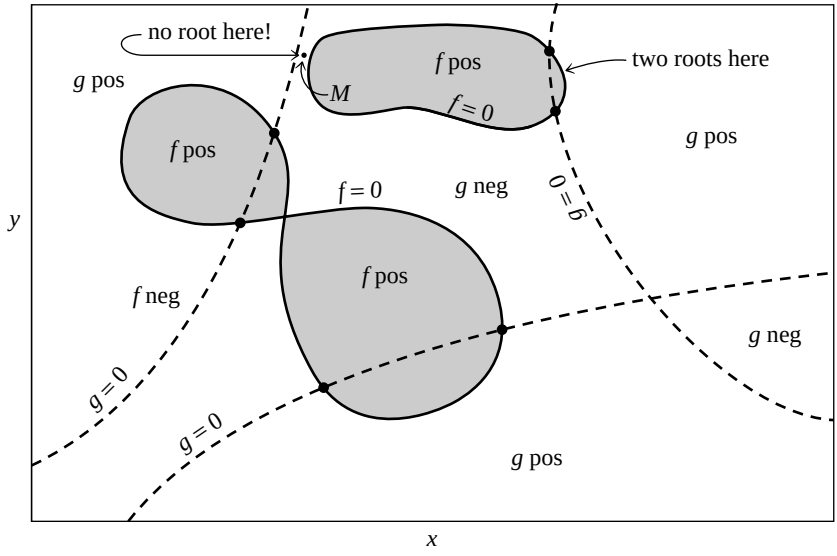


Figure 9.6.1. Solution of two nonlinear equations in two unknowns. Solid curves refer to $f(x,y)$, dashed curves to $g(x,y)$. Each equation divides the (x,y) plane into positive and negative regions, bounded by zero curves. The desired solutions are the intersections of these unrelated zero curves. The number of solutions is *a priori* unknown.

You see that root finding becomes virtually impossible without insight! You will almost always have to use additional information, specific to your particular problem, to answer such basic questions as, "Do I expect a unique solution?" and "Approximately where?" Acton [1] has a good discussion of some of the particular strategies that can be tried.

In this section we will discuss the simplest multidimensional root finding method, Newton-Raphson. This method gives you a very efficient means of converging to a root, if you have a sufficiently good initial guess. It can also spectacularly fail to converge, indicating (though not proving) that your putative root does not exist nearby. In §9.7 we discuss more sophisticated implementations of the Newton-Raphson method, which try to improve on Newton-Raphson's poor global convergence. A multidimensional generalization of the secant method, called Broyden's method, is also discussed in §9.7.

A typical problem gives N functional relations to be zeroed, involving variables $x_i, i = 1, 2, \ldots, N$:

$$F_i(x_1, x_2, \ldots, x_N) = 0 \qquad i = 1, 2, \ldots, N. \tag{9.6.2}$$

We let $\mathbf{x}$ denote the entire vector of values x_i and $\mathbf{F}$ denote the entire vector of functions F_i. In the neighborhood of $\mathbf{x}$, each of the functions F_i can be expanded in Taylor series

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2). \tag{9.6.3}$$

The matrix of partial derivatives appearing in equation (9.6.3) is the *Jacobian* matrix **J**:

$$J_{ij} \equiv \frac{\partial F_i}{\partial x_j}. \tag{9.6.4}$$

In matrix notation equation (9.6.3) is

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2). \tag{9.6.5}$$

By neglecting terms of order $\delta\mathbf{x}^2$ and higher and by setting $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$, we obtain a set of linear equations for the corrections $\delta\mathbf{x}$ that move each function closer to zero simultaneously, namely

$$\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F}. \tag{9.6.6}$$

Matrix equation (9.6.6) can be solved by LU decomposition as described in §2.3. The corrections are then added to the solution vector,

$$\mathbf{x}_{\rm new} = \mathbf{x}_{\rm old} + \delta\mathbf{x} \tag{9.6.7}$$

and the process is iterated to convergence. In general it is a good idea to check the degree to which both functions and variables have converged. Once either reaches machine accuracy, the other won't change.

The following routine `mnewt` performs `ntrial` iterations starting from an initial guess at the solution vector `x` of length `n` variables. Iteration stops if either the sum of the magnitudes of the functions F_i is less than some tolerance `tolf`, or the sum of the absolute values of the corrections to δx_i is less than some tolerance `tolx`. `mnewt` calls a user supplied subroutine `usrfun` which must return the function values **F** and the Jacobian matrix **J**. If **J** is difficult to compute analytically, you can try having `usrfun` call the routine `fdjac` of §9.7 to compute the partial derivatives by finite differences. You should not make `ntrial` too big; rather inspect to see what is happening before continuing for some further iterations.

```
      SUBROUTINE mnewt(ntrial,x,n,tolx,tolf)
      INTEGER n,ntrial,NP
      REAL tolf,tolx,x(n)
      PARAMETER (NP=15)                       Up to NP variables.
C     USES lubksb,ludcmp,usrfun
        Given an initial guess x for a root in n dimensions, take ntrial Newton-Raphson steps to
        improve the root. Stop if the root converges in either summed absolute variable increments
        tolx or summed absolute function values tolf.
      INTEGER i,k,indx(NP)
      REAL d,errf,errx,fjac(NP,NP),fvec(NP),p(NP)
      do 14 k=1,ntrial
        call usrfun(x,n,NP,fvec,fjac)         User subroutine supplies function values at x in fvec
        errf=0.                                   and Jacobian matrix in fjac.
        do 11 i=1,n                           Check function convergence.
          errf=errf+abs(fvec(i))
        enddo 11
        if(errf.le.tolf)return
        do 12 i=1,n                           Right-hand side of linear equations.
          p(i)=-fvec(i)
        enddo 12
```

```
    call ludcmp(fjac,n,NP,indx,d)    Solve linear equations using LU decomposition.
    call lubksb(fjac,n,NP,indx,p)
    errx=0.                          Check root convergence.
    do 13 i=1,n                      Update solution.
        errx=errx+abs(p(i))
        x(i)=x(i)+p(i)
    enddo 13
    if(errx.le.tolx)return
enddo 14
return
END
```

Newton's Method versus Minimization

In the next chapter, we will find that there *are* efficient general techniques for finding a minimum of a function of many variables. Why is that task (relatively) easy, while multidimensional root finding is often quite hard? Isn't minimization equivalent to finding a zero of an N-dimensional gradient vector, not so different from zeroing an N-dimensional function? No! The components of a gradient vector are not independent, arbitrary functions. Rather, they obey so-called integrability conditions that are highly restrictive. Put crudely, you can always find a minimum by sliding downhill on a single surface. The test of "downhillness" is thus one-dimensional. There is no analogous conceptual procedure for finding a multidimensional root, where "downhill" must mean simultaneously downhill in N separate function spaces, thus allowing a multitude of trade-offs, as to how much progress in one dimension is worth compared with progress in another.

It might occur to you to carry out multidimensional root finding by collapsing all these dimensions into one: Add up the sums of squares of the individual functions F_i to get a master function F which (i) is positive definite, and (ii) has a global minimum of zero exactly at all solutions of the original set of nonlinear equations. Unfortunately, as you will see in the next chapter, the efficient algorithms for finding minima come to rest on global and local minima indiscriminately. You will often find, to your great dissatisfaction, that your function F has a great number of local minima. In Figure 9.6.1, for example, there is likely to be a local minimum wherever the zero contours of f and g make a close approach to each other. The point labeled M is such a point, and one sees that there are no nearby roots.

However, we will now see that sophisticated strategies for multidimensional root finding can in fact make use of the idea of minimizing a master function F, by *combining* it with Newton's method applied to the full set of functions F_i. While such methods can still occasionally fail by coming to rest on a local minimum of F, they often succeed where a direct attack via Newton's method alone fails. The next section deals with these methods.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 14. [1]

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press).

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).

9.7 Globally Convergent Methods for Nonlinear Systems of Equations

We have seen that Newton's method for solving nonlinear equations has an unfortunate tendency to wander off into the wild blue yonder if the initial guess is not sufficiently close to the root. A *global* method is one that converges to a solution from almost any starting point. In this section we will develop an algorithm that combines the rapid local convergence of Newton's method with a globally convergent strategy that will guarantee some progress towards the solution at each iteration. The algorithm is closely related to the quasi-Newton method of minimization which we will describe in §10.7.

Recall our discussion of §9.6: the Newton step for the set of equations

$$\mathbf{F}(\mathbf{x}) = 0 \tag{9.7.1}$$

is

$$\mathbf{x}_{\rm new} = \mathbf{x}_{\rm old} + \delta\mathbf{x} \tag{9.7.2}$$

where

$$\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \tag{9.7.3}$$

Here $\mathbf{J}$ is the Jacobian matrix. How do we decide whether to accept the Newton step $\delta\mathbf{x}$? A reasonable strategy is to require that the step decrease $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$. This is the same requirement we would impose if we were trying to minimize

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F} \tag{9.7.4}$$

(The $\frac{1}{2}$ is for later convenience.) Every solution to (9.7.1) minimizes (9.7.4), but there may be local minima of (9.7.4) that are not solutions to (9.7.1). Thus, as already mentioned, simply applying one of our minimum finding algorithms from Chapter 10 to (9.7.4) is *not* a good idea.

To develop a better strategy, note that the Newton step (9.7.3) is a *descent direction* for f:

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0 \tag{9.7.5}$$

Thus our strategy is quite simple: We always first try the full Newton step, because once we are close enough to the solution we will get quadratic convergence. However, we check at each iteration that the proposed step reduces f. If not, we *backtrack* along the Newton direction until we have an acceptable step. Because the Newton step is a descent direction for f, we are guaranteed to find an acceptable step by backtracking. We will discuss the backtracking algorithm in more detail below.

Note that this method essentially minimizes f by taking Newton steps designed to bring $\mathbf{F}$ to zero. This is *not* equivalent to minimizing f directly by taking Newton steps designed to bring ∇f to zero. While the method can still occasionally fail by landing on a local minimum of f, this is quite rare in practice. The routine `newt` below will warn you if this happens. The remedy is to try a new starting point.

9.7 Globally Convergent Methods for Nonlinear Systems of Equations

We have seen that Newton's method for solving nonlinear equations has an unfortunate tendency to wander off into the wild blue yonder if the initial guess is not sufficiently close to the root. A *global* method is one that converges to a solution from almost any starting point. In this section we will develop an algorithm that combines the rapid local convergence of Newton's method with a globally convergent strategy that will guarantee some progress towards the solution at each iteration. The algorithm is closely related to the quasi-Newton method of minimization which we will describe in §10.7.

Recall our discussion of §9.6: the Newton step for the set of equations

$$\mathbf{F}(\mathbf{x}) = 0 \tag{9.7.1}$$

is

$$\mathbf{x}_{\rm new} = \mathbf{x}_{\rm old} + \delta\mathbf{x} \tag{9.7.2}$$

where

$$\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \tag{9.7.3}$$

Here $\mathbf{J}$ is the Jacobian matrix. How do we decide whether to accept the Newton step $\delta\mathbf{x}$? A reasonable strategy is to require that the step decrease $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$. This is the same requirement we would impose if we were trying to minimize

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F} \tag{9.7.4}$$

(The $\frac{1}{2}$ is for later convenience.) Every solution to (9.7.1) minimizes (9.7.4), but there may be local minima of (9.7.4) that are not solutions to (9.7.1). Thus, as already mentioned, simply applying one of our minimum finding algorithms from Chapter 10 to (9.7.4) is *not* a good idea.

To develop a better strategy, note that the Newton step (9.7.3) is a *descent direction* for f:

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0 \tag{9.7.5}$$

Thus our strategy is quite simple: We always first try the full Newton step, because once we are close enough to the solution we will get quadratic convergence. However, we check at each iteration that the proposed step reduces f. If not, we *backtrack* along the Newton direction until we have an acceptable step. Because the Newton step is a descent direction for f, we are guaranteed to find an acceptable step by backtracking. We will discuss the backtracking algorithm in more detail below.

Note that this method essentially minimizes f by taking Newton steps designed to bring $\mathbf{F}$ to zero. This is *not* equivalent to minimizing f directly by taking Newton steps designed to bring ∇f to zero. While the method can still occasionally fail by landing on a local minimum of f, this is quite rare in practice. The routine `newt` below will warn you if this happens. The remedy is to try a new starting point.

Line Searches and Backtracking

When we are not close enough to the minimum of f, taking the full Newton step $\mathbf{p} = \delta\mathbf{x}$ need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. So the goal is to move to a new point $\mathbf{x}_{\rm new}$ along the *direction* of the Newton step $\mathbf{p}$, but not necessarily all the way:

$$\mathbf{x}_{\rm new} = \mathbf{x}_{\rm old} + \lambda\mathbf{p}, \qquad 0 < \lambda \leq 1 \tag{9.7.6}$$

The aim is to find λ so that $f(\mathbf{x}_{\rm old} + \lambda\mathbf{p})$ has decreased sufficiently. Until the early 1970s, standard practice was to choose λ so that $\mathbf{x}_{\rm new}$ exactly minimizes f in the direction $\mathbf{p}$. However, we now know that it is extremely wasteful of function evaluations to do so. A better strategy is as follows: Since $\mathbf{p}$ is always the Newton direction in our algorithms, we first try $\lambda = 1$, the full Newton step. This will lead to quadratic convergence when $\mathbf{x}$ is sufficiently close to the solution. However, if $f(\mathbf{x}_{\rm new})$ does not meet our acceptance criteria, we *backtrack* along the Newton direction, trying a smaller value of λ, until we find a suitable point. Since the Newton direction is a descent direction, we are guaranteed to decrease f for sufficiently small λ.

What should the criterion for accepting a step be? It is *not* sufficient to require merely that $f(\mathbf{x}_{\rm new}) < f(\mathbf{x}_{\rm old})$. This criterion can fail to converge to a minimum of f in one of two ways. First, it is possible to construct a sequence of steps satisfying this criterion with f decreasing too slowly relative to the step lengths. Second, one can have a sequence where the step lengths are too small relative to the initial rate of decrease of f. (For examples of such sequences, see [1], p. 117.)

A simple way to fix the first problem is to require the *average* rate of decrease of f to be at least some fraction α of the *initial* rate of decrease $\nabla f \cdot \mathbf{p}$:

$$f(\mathbf{x}_{\rm new}) \leq f(\mathbf{x}_{\rm old}) + \alpha\nabla f \cdot (\mathbf{x}_{\rm new} - \mathbf{x}_{\rm old}) \tag{9.7.7}$$

Here the parameter α satisfies $0 < \alpha < 1$. We can get away with quite small values of α; $\alpha = 10^{-4}$ is a good choice.

The second problem can be fixed by requiring the rate of decrease of f at $\mathbf{x}_{\rm new}$ to be greater than some fraction β of the rate of decrease of f at $\mathbf{x}_{\rm old}$. In practice, we will not need to impose this second constraint because our backtracking algorithm will have a built-in cutoff to avoid taking steps that are too small.

Here is the strategy for a practical backtracking routine: Define

$$g(\lambda) \equiv f(\mathbf{x}_{\rm old} + \lambda\mathbf{p}) \tag{9.7.8}$$

so that

$$g'(\lambda) = \nabla f \cdot \mathbf{p} \tag{9.7.9}$$

If we need to backtrack, then we model g with the most current information we have and choose λ to minimize the model. We start with $g(0)$ and $g'(0)$ available. The first step is always the Newton step, $\lambda = 1$. If this step is not acceptable, we have available $g(1)$ as well. We can therefore model $g(\lambda)$ as a quadratic:

$$g(\lambda) \approx [g(1) - g(0) - g'(0)]\lambda^2 + g'(0)\lambda + g(0) \tag{9.7.10}$$

Taking the derivative of this quadratic, we find that it is a minimum when

$$\lambda = -\frac{g'(0)}{2[g(1) - g(0) - g'(0)]} \tag{9.7.11}$$

Since the Newton step failed, we can show that $\lambda \lesssim \frac{1}{2}$ for small α. We need to guard against too small a value of λ, however. We set $\lambda_{\rm min} = 0.1$.

On second and subsequent backtracks, we model g as a cubic in λ, using the previous value $g(\lambda_1)$ and the second most recent value $g(\lambda_2)$:

$$g(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0) \tag{9.7.12}$$

Requiring this expression to give the correct values of g at λ_1 and λ_2 gives two equations that can be solved for the coefficients a and b:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1/\lambda_1^2 & -1/\lambda_2^2 \\ -\lambda_2/\lambda_1^2 & \lambda_1/\lambda_2^2 \end{bmatrix} \cdot \begin{bmatrix} g(\lambda_1) - g'(0)\lambda_1 - g(0) \\ g(\lambda_2) - g'(0)\lambda_2 - g(0) \end{bmatrix} \tag{9.7.13}$$

The minimum of the cubic (9.7.12) is at

$$\lambda = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a} \tag{9.7.14}$$

We enforce that λ lie between $\lambda_{\max} = 0.5\lambda_1$ and $\lambda_{\min} = 0.1\lambda_1$.

The routine has two additional features, a minimum step length `alamin` and a maximum step length `stpmax`. `lnsrch` will also be used in the quasi-Newton minimization routine `dfpmin` in the next section.

```
      SUBROUTINE lnsrch(n,xold,fold,g,p,x,f,stpmax,check,func)
      INTEGER n
      LOGICAL check
      REAL f,fold,stpmax,g(n),p(n),x(n),xold(n),func,ALF,TOLX
      PARAMETER (ALF=1.e-4,TOLX=1.e-7)
      EXTERNAL func
C     USES func
        Given an n-dimensional point xold(1:n), the value of the function and gradient there,
        fold and g(1:n), and a direction p(1:n), finds a new point x(1:n) along the direction
        p from xold where the function func has decreased "sufficiently." The new function value
        is returned in f. stpmax is an input quantity that limits the length of the steps so that you
        do not try to evaluate the function in regions where it is undefined or subject to overflow.
        p is usually the Newton direction. The output quantity check is false on a normal exit.
        It is true when x is too close to xold. In a minimization algorithm, this usually signals
        convergence and can be ignored. However, in a zero-finding algorithm the calling program
        should check whether the convergence is spurious.
        Parameters: ALF ensures sufficient decrease in function value; TOLX is the convergence
        criterion on Δx.
      INTEGER i
      REAL a,alam,alam2,alamin,b,disc,f2,rhs1,rhs2,slope,
     *     sum,temp,test,tmplam
      check=.false.
      sum=0.
      do 11 i=1,n
        sum=sum+p(i)*p(i)
      enddo 11
      sum=sqrt(sum)
      if(sum.gt.stpmax)then                    Scale if attempted step is too big.
        do 12 i=1,n
          p(i)=p(i)*stpmax/sum
        enddo 12
      endif
      slope=0.
      do 13 i=1,n
        slope=slope+g(i)*p(i)
      enddo 13
      if(slope.ge.0.) pause 'roundoff problem in lnsrch'
      test=0.                                  Compute λmin.
      do 14 i=1,n
        temp=abs(p(i))/max(abs(xold(i)),1.)
        if(temp.gt.test)test=temp
      enddo 14
      alamin=TOLX/test
      alam=1.                                  Always try full Newton step first.
1     continue                                 Start of iteration loop.
        do 15 i=1,n
          x(i)=xold(i)+alam*p(i)
        enddo 15
```

```
      f=func(x)
      if(alam.lt.alamin)then                    Convergence on Δx. For zero finding,
          do 16 i=1,n                               the calling program should verify the
              x(i)=xold(i)                          convergence.
          enddo 16
          check=.true.
          return
      else if(f.le.fold+ALF*alam*slope)then     Sufficient function decrease.
          return
      else                                      Backtrack.
          if(alam.eq.1.)then                    First time.
              tmplam=-slope/(2.*(f-fold-slope))
          else                                  Subsequent backtracks.
              rhs1=f-fold-alam*slope
              rhs2=f2-fold-alam2*slope
              a=(rhs1/alam**2-rhs2/alam2**2)/(alam-alam2)
              b=(-alam2*rhs1/alam**2+alam*rhs2/alam2**2)/
     *             (alam-alam2)
              if(a.eq.0.)then
                  tmplam=-slope/(2.*b)
              else
                  disc=b*b-3.*a*slope
                  if(disc.lt.0.)then
                      tmplam=.5*alam
                  else if(b.le.0.)then
                      tmplam=(-b+sqrt(disc))/(3.*a)
                  else
                      tmplam=-slope/(b+sqrt(disc))
                  endif
              endif
              if(tmplam.gt..5*alam)tmplam=.5*alam      λ ≤ 0.5λ1.
          endif
      endif
      alam2=alam
      f2=f
      alam=max(tmplam,.1*alam)                  λ ≥ 0.1λ1.
    goto 1                                      Try again.
    END
```

Here now is the globally convergent Newton routine `newt` that uses `lnsrch`. A feature of `newt` is that you need not supply the Jacobian matrix analytically; the routine will attempt to compute the necessary partial derivatives of **F** by finite differences in the routine `fdjac`. This routine uses some of the techniques described in §5.7 for computing numerical derivatives. Of course, you can always replace `fdjac` with a routine that calculates the Jacobian analytically if this is easy for you to do.

```
    SUBROUTINE newt(x,n,check)
    INTEGER n,nn,NP,MAXITS
    LOGICAL check
    REAL x(n),fvec,TOLF,TOLMIN,TOLX,STPMX
    PARAMETER (NP=40,MAXITS=200,TOLF=1.e-4,TOLMIN=1.e-6,TOLX=1.e-7,
   *     STPMX=100.)
    COMMON /newtv/ fvec(NP),nn              Communicates with fmin.
    SAVE /newtv/
C   USES fdjac,fmin,lnsrch,lubksb,ludcmp
```

Given an initial guess `x(1:n)` for a root in `n` dimensions, find the root by a globally convergent Newton's method. The vector of functions to be zeroed, called `fvec(1:n)` in the routine below, is returned by a user-supplied subroutine that *must* be called `funcv` and have the declaration `subroutine funcv(n,x,fvec)`. The output quantity `check` is false on a normal return and true if the routine has converged to a local minimum of the function `fmin` defined below. In this case try restarting from a different initial guess.

Parameters: `NP` is the maximum expected value of `n`; `MAXITS` is the maximum number of iterations; `TOLF` sets the convergence criterion on function values; `TOLMIN` sets the criterion for deciding whether spurious convergence to a minimum of `fmin` has occurred; `TOLX` is

the convergence criterion on $\delta\mathbf{x}$; STPMX is the scaled maximum step length allowed in line searches.

```
      INTEGER i,its,j,indx(NP)
      REAL d,den,f,fold,stpmax,sum,temp,test,fjac(NP,NP),
     *     g(NP),p(NP),xold(NP),fmin
      EXTERNAL fmin
      nn=n
      f=fmin(x)                          The vector fvec is also computed by this call.
      test=0.                            Test for initial guess being a root. Use more stringent test than simply TOLF.
      do 11 i=1,n
        if(abs(fvec(i)).gt.test)test=abs(fvec(i))
      enddo 11
      if(test.lt..01*TOLF)then
        check=.false.
        return
      endif
      sum=0.                             Calculate stpmax for line searches.
      do 12 i=1,n
        sum=sum+x(i)**2
      enddo 12
      stpmax=STPMX*max(sqrt(sum),float(n))
      do 21 its=1,MAXITS                 Start of iteration loop.
        call fdjac(n,x,fvec,NP,fjac)
          If analytic Jacobian is available, you can replace the routine fdjac below with your own routine.
        do 14 i=1,n                      Compute $\nabla f$ for the line search.
          sum=0.
          do 13 j=1,n
            sum=sum+fjac(j,i)*fvec(j)
          enddo 13
          g(i)=sum
        enddo 14
        do 15 i=1,n                      Store $\mathbf{x}$,
          xold(i)=x(i)
        enddo 15
        fold=f                           and $f$.
        do 16 i=1,n                      Right-hand side for linear equations.
          p(i)=-fvec(i)
        enddo 16
        call ludcmp(fjac,n,NP,indx,d)    Solve linear equations by $LU$ decomposition.
        call lubksb(fjac,n,NP,indx,p)
        call lnsrch(n,xold,fold,g,p,x,f,stpmax,check,fmin)
          lnsrch returns new $\mathbf{x}$ and $f$. It also calculates fvec at the new $\mathbf{x}$ when it calls fmin.
        test=0.                          Test for convergence on function values.
        do 17 i=1,n
          if(abs(fvec(i)).gt.test)test=abs(fvec(i))
        enddo 17
        if(test.lt.TOLF)then
          check=.false.
          return
        endif
        if(check)then                    Check for gradient of $f$ zero, i.e., spurious convergence.
          test=0.
          den=max(f,.5*n)
          do 18 i=1,n
            temp=abs(g(i))*max(abs(x(i)),1.)/den
            if(temp.gt.test)test=temp
          enddo 18
          if(test.lt.TOLMIN)then
            check=.true.
          else
            check=.false.
          endif
          return
```

```
        endif
        test=0.                                  Test for convergence on δx.
        do 19 i=1,n
            temp=(abs(x(i)-xold(i)))/max(abs(x(i)),1.)
            if(temp.gt.test)test=temp
        enddo 19
        if(test.lt.TOLX)return
    enddo 21
    pause 'MAXITS exceeded in newt'
    END

    SUBROUTINE fdjac(n,x,fvec,np,df)
    INTEGER n,np,NMAX
    REAL df(np,np),fvec(n),x(n),EPS
    PARAMETER (NMAX=40,EPS=1.e-4)
C   USES funcv
```

Computes forward-difference approximation to Jacobian. On input, `x(1:n)` is the point at which the Jacobian is to be evaluated, `fvec(1:n)` is the vector of function values at the point, and `np` is the physical dimension of the Jacobian array `df(1:n,1:n)` which is output. `subroutine funcv(n,x,f)` is a fixed-name, user-supplied routine that returns the vector of functions at `x`.

Parameters: `NMAX` is the maximum value of `n`; `EPS` is the approximate square root of the machine precision.

```
    INTEGER i,j
    REAL h,temp,f(NMAX)
    do 12 j=1,n
        temp=x(j)
        h=EPS*abs(temp)
        if(h.eq.0.)h=EPS
        x(j)=temp+h                              Trick to reduce finite precision error.
        h=x(j)-temp
        call funcv(n,x,f)
        x(j)=temp
        do 11 i=1,n                              Forward difference formula.
            df(i,j)=(f(i)-fvec(i))/h
        enddo 11
    enddo 12
    return
    END

    FUNCTION fmin(x)
    INTEGER n,NP
    REAL fmin,x(*),fvec
    PARAMETER (NP=40)
    COMMON /newtv/ fvec(NP),n
    SAVE /newtv/
C   USES funcv
```

Returns $f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F}$ at `x`. `subroutine funcv(n,x,f)` is a fixed-name, user-supplied routine that returns the vector of functions at `x`. The common block `newtv` communicates the function values back to `newt`.

```
    INTEGER i
    REAL sum
    call funcv(n,x,fvec)
    sum=0.
    do 11 i=1,n
        sum=sum+fvec(i)**2
    enddo 11
    fmin=0.5*sum
    return
    END
```

The routine `newt` assumes that typical values of all components of $\mathbf{x}$ and of $\mathbf{F}$ are of order unity, and it can fail if this assumption is badly violated. You should rescale the variables by their typical values before invoking `newt` if this problem occurs.

Multidimensional Secant Methods: Broyden's Method

Newton's method as implemented above is quite powerful, but it still has several disadvantages. One drawback is that the Jacobian matrix is needed. In many problems analytic derivatives are unavailable. If function evaluation is expensive, then the cost of finite-difference determination of the Jacobian can be prohibitive.

Just as the quasi-Newton methods to be discussed in §10.7 provide cheap approximations for the Hessian matrix in minimization algorithms, there are quasi-Newton methods that provide cheap approximations to the Jacobian for zero finding. These methods are often called *secant methods*, since they reduce to the secant method (§9.2) in one dimension (see, e.g., [1]). The best of these methods still seems to be the first one introduced, *Broyden's method* [2].

Let us denote the approximate Jacobian by $\mathbf{B}$. Then the ith quasi-Newton step $\delta\mathbf{x}_i$ is the solution of

$$\mathbf{B}_i \cdot \delta\mathbf{x}_i = -\mathbf{F}_i \tag{9.7.15}$$

where $\delta\mathbf{x}_i = \mathbf{x}_{i+1} - \mathbf{x}_i$ (cf. equation 9.7.3). The quasi-Newton or secant condition is that $\mathbf{B}_{i+1}$ satisfy

$$\mathbf{B}_{i+1} \cdot \delta\mathbf{x}_i = \delta\mathbf{F}_i \tag{9.7.16}$$

where $\delta\mathbf{F}_i = \mathbf{F}_{i+1} - \mathbf{F}_i$. This is the generalization of the one-dimensional secant approximation to the derivative, $\delta F/\delta x$. However, equation (9.7.16) does not determine $\mathbf{B}_{i+1}$ uniquely in more than one dimension.

Many different auxiliary conditions to pin down $\mathbf{B}_{i+1}$ have been explored, but the best-performing algorithm in practice results from Broyden's formula. This formula is based on the idea of getting $\mathbf{B}_{i+1}$ by making the least change to $\mathbf{B}_i$ consistent with the secant equation (9.7.16). Broyden showed that the resulting formula is

$$\mathbf{B}_{i+1} = \mathbf{B}_i + \frac{(\delta\mathbf{F}_i - \mathbf{B}_i \cdot \delta\mathbf{x}_i) \otimes \delta\mathbf{x}_i}{\delta\mathbf{x}_i \cdot \delta\mathbf{x}_i} \tag{9.7.17}$$

You can easily check that $\mathbf{B}_{i+1}$ satisfies (9.7.16).

Early implementations of Broyden's method used the Sherman-Morrison formula, equation (2.7.2), to invert equation (9.7.17) analytically,

$$\mathbf{B}_{i+1}^{-1} = \mathbf{B}_i^{-1} + \frac{(\delta\mathbf{x}_i - \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i) \otimes \delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1}}{\delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i} \tag{9.7.18}$$

Then instead of solving equation (9.7.3) by e.g., LU decomposition, one determined

$$\delta\mathbf{x}_i = -\mathbf{B}_i^{-1} \cdot \mathbf{F}_i \tag{9.7.19}$$

by matrix multiplication in $O(N^2)$ operations. The disadvantage of this method is that it cannot easily be embedded in a globally convergent strategy, for which the gradient of equation (9.7.4) requires $\mathbf{B}$, not $\mathbf{B}^{-1}$,

$$\nabla(\tfrac{1}{2}\mathbf{F} \cdot \mathbf{F}) \simeq \mathbf{B}^T \cdot \mathbf{F} \tag{9.7.20}$$

Accordingly, we implement the update formula in the form (9.7.17).

However, we can still preserve the $O(N^2)$ solution of (9.7.3) by using QR decomposition (§2.10) instead of LU decomposition. The reason is that because of the special form of equation (9.7.17), the QR decomposition of $\mathbf{B}_i$ can be updated into the QR decomposition of $\mathbf{B}_{i+1}$ in $O(N^2)$ operations (§2.10). All we need is an initial approximation $\mathbf{B}_0$ to start the ball rolling. It is often acceptable to start simply with the identity matrix, and then allow $O(N)$ updates to produce a reasonable approximation to the Jacobian. We prefer to spend the first N function evaluations on a finite-difference approximation to initialize $\mathbf{B}$ via a call to `fdjac`.

Since $\mathbf{B}$ is not the exact Jacobian, we are not guaranteed that $\delta\mathbf{x}$ is a descent direction for $f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F}$ (cf. equation 9.7.5). Thus the line search algorithm can fail to return a suitable step if $\mathbf{B}$ wanders far from the true Jacobian. In this case, we reinitialize $\mathbf{B}$ by another call to `fdjac`.

Like the secant method in one dimension, Broyden's method converges superlinearly once you get close enough to the root. Embedded in a global strategy, it is almost as robust

as Newton's method, and often needs far fewer function evaluations to determine a zero. Note that the final value of **B** is *not* always close to the true Jacobian at the root, even when the method converges.

The routine broydn given below is very similar to newt in organization. The principal differences are the use of QR decomposition instead of LU, and the updating formula instead of directly determining the Jacobian. The remarks at the end of newt about scaling the variables apply equally to broydn.

```
      SUBROUTINE broydn(x,n,check)
      INTEGER n,nn,NP,MAXITS
      REAL x(n),fvec,EPS,TOLF,TOLMIN,TOLX,STPMX
      LOGICAL check
      PARAMETER (NP=40,MAXITS=200,EPS=1.e-7,TOLF=1.e-4,TOLMIN=1.e-6,
     *     TOLX=EPS,STPMX=100.)
      COMMON /newtv/ fvec(NP),nn                Communicates with fmin.
      SAVE /newtv/
C     USES fdjac,fmin,lnsrch,qrdcmp,qrupdt,rsolv
        Given an initial guess x(1:n) for a root in n dimensions, find the root by Broyden's method
        embedded in a globally convergent strategy. The vector of functions to be zeroed, called
        fvec(1:n) in the routine below, is returned by a user-supplied subroutine that must be
        called funcv and have the declaration subroutine funcv(n,x,fvec). The subroutine
        fdjac and the function fmin from newt are used. The output quantity check is false on
        a normal return and true if the routine has converged to a local minimum of the function
        fmin or if Broyden's method can make no further progress. In this case try restarting from
        a different initial guess.
        Parameters: NP is the maximum expected value of n; MAXITS is the maximum number of
        iterations; EPS is close to the machine precision; TOLF sets the convergence criterion on
        function values; TOLMIN sets the criterion for deciding whether spurious convergence to a
        minimum of fmin has occurred; TOLX is the convergence criterion on δx; STPMX is the
        scaled maximum step length allowed in line searches.
      INTEGER i,its,j,k
      REAL den,f,fold,stpmax,sum,temp,test,c(NP),d(NP),fvcold(NP),
     *     g(NP),p(NP),qt(NP,NP),r(NP,NP),s(NP),t(NP),w(NP),
     *     xold(NP),fmin
      LOGICAL restrt,sing,skip
      EXTERNAL fmin
      nn=n
      f=fmin(x)                                 The vector fvec is also computed by this call.
      test=0.                                   Test for initial guess being a root. Use more strin-
      do 11 i=1,n                                   gent test than simply TOLF.
        if(abs(fvec(i)).gt.test)test=abs(fvec(i))
      enddo 11
      if(test.lt..01*TOLF)then
        check=.false.
        return
      endif
      sum=0.                                    Calculate stpmax for line searches.
      do 12 i=1,n
        sum=sum+x(i)**2
      enddo 12
      stpmax=STPMX*max(sqrt(sum),float(n))
      restrt=.true.                             Ensure initial Jacobian gets computed.
      do 44 its=1,MAXITS                        Start of iteration loop.
        if(restrt)then
          call fdjac(n,x,fvec,NP,r)             Initialize or reinitialize Jacobian in r.
          call qrdcmp(r,n,NP,c,d,sing)          QR decomposition of Jacobian.
          if(sing) pause 'singular Jacobian in broydn'
          do 14 i=1,n                           Form Q^T explicitly.
            do 13 j=1,n
              qt(i,j)=0.
            enddo 13
            qt(i,i)=1.
          enddo 14
```

```
        do 18 k=1,n-1
            if(c(k).ne.0.)then
                do 17 j=1,n
                    sum=0.
                    do 15 i=k,n
                        sum=sum+r(i,k)*qt(i,j)
                    enddo 15
                    sum=sum/c(k)
                    do 16 i=k,n
                        qt(i,j)=qt(i,j)-sum*r(i,k)
                    enddo 16
                enddo 17
            endif
        enddo 18
        do 21 i=1,n                          Form R explicitly.
            r(i,i)=d(i)
            do 19 j=1,i-1
                r(i,j)=0.
            enddo 19
        enddo 21
    else                                     Carry out Broyden update.
        do 22 i=1,n                          s = δx.
            s(i)=x(i)-xold(i)
        enddo 22
        do 24 i=1,n                          t = R · s.
            sum=0.
            do 23 j=i,n
                sum=sum+r(i,j)*s(j)
            enddo 23
            t(i)=sum
        enddo 24
        skip=.true.
        do 26 i=1,n                          w = δF − B · s.
            sum=0.
            do 25 j=1,n
                sum=sum+qt(j,i)*t(j)
            enddo 25
            w(i)=fvec(i)-fvcold(i)-sum
            if(abs(w(i)).ge.EPS*(abs(fvec(i))+abs(fvcold(i))))then
              Don't update with noisy components of w.
                skip=.false.
            else
                w(i)=0.
            endif
        enddo 26
        if(.not.skip)then
            do 28 i=1,n                      t = Q^T · w.
                sum=0.
                do 27 j=1,n
                    sum=sum+qt(i,j)*w(j)
                enddo 27
                t(i)=sum
            enddo 28
            den=0.
            do 29 i=1,n
                den=den+s(i)**2
            enddo 29
            do 31 i=1,n                      Store s/(s · s) in s.
                s(i)=s(i)/den
            enddo 31
            call qrupdt(r,qt,n,NP,t,s)       Update R and Q^T.
            do 32 i=1,n
                if(r(i,i).eq.0.) pause 'r singular in broydn'
                d(i)=r(i,i)                  Diagonal of R stored in d.
```

```
        enddo 32
    endif
endif
do 34 i=1,n                          Compute ∇f ≈ (Q · R)^T · F for the line search.
    sum=0.
    do 33 j=1,n
        sum=sum+qt(i,j)*fvec(j)
    enddo 33
    g(i)=sum
enddo 34
do 36 i=n,1,-1
    sum=0.
    do 35 j=1,i
        sum=sum+r(j,i)*g(j)
    enddo 35
    g(i)=sum
enddo 36
do 37 i=1,n                          Store x and F.
    xold(i)=x(i)
    fvcold(i)=fvec(i)
enddo 37
fold=f                               Store f.
do 39 i=1,n                          Right-hand side for linear equations is −Q^T · F.
    sum=0.
    do 38 j=1,n
        sum=sum+qt(i,j)*fvec(j)
    enddo 38
    p(i)=-sum
enddo 39
call rsolv(r,n,NP,d,p)               Solve linear equations.
call lnsrch(n,xold,fold,g,p,x,f,stpmax,check,fmin)
  lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
test=0.                              Test for convergence on function values.
do 41 i=1,n
    if(abs(fvec(i)).gt.test)test=abs(fvec(i))
enddo 41
if(test.lt.TOLF)then
    check=.false.
    return
endif
if(check)then                        True if line search failed to find a new x.
    if(restrt)then                   Failure; already tried reinitializing the Jacobian.
        return
    else                             Check for gradient of f zero, i.e., spurious con-
        test=0.                          vergence.
        den=max(f,.5*n)
        do 42 i=1,n
            temp=abs(g(i))*max(abs(x(i)),1.)/den
            if(temp.gt.test)test=temp
        enddo 42
        if(test.lt.TOLMIN)then
            return
        else                         Try reinitializing the Jacobian.
            restrt=.true.
        endif
    endif
else                                 Successful step; will use Broyden update for next
    restrt=.false.                       step.
    test=0.                          Test for convergence on δx.
    do 43 i=1,n
        temp=(abs(x(i)-xold(i)))/max(abs(x(i)),1.)
        if(temp.gt.test)test=temp
    enddo 43
    if(test.lt.TOLX)return
```

```
    endif
enddo 44
pause 'MAXITS exceeded in broydn'
END
```

More Advanced Implementations

One of the principal ways that the methods described so far can fail is if **J** (in Newton's method) or **B** in (Broyden's method) becomes singular or nearly singular, so that $\delta\mathbf{x}$ cannot be determined. If you are lucky, this situation will not occur very often in practice. Methods developed so far to deal with this problem involve monitoring the condition number of **J** and perturbing **J** if singularity or near singularity is detected. This is most easily implemented if the QR decomposition is used instead of LU in Newton's method (see [1] for details). Our personal experience is that, while such an algorithm can solve problems where **J** is exactly singular and the standard Newton's method fails, it is occasionally less robust on other problems where LU decomposition succeeds. Clearly implementation details involving roundoff, underflow, etc., are important here and the last word is yet to be written.

Our global strategies both for minimization and zero finding have been based on line searches. Other global algorithms, such as the *hook step* and *dogleg step* methods, are based instead on the *model-trust region* approach, which is related to the Levenberg-Marquardt algorithm for nonlinear least-squares (§15.5). While somewhat more complicated than line searches, these methods have a reputation for robustness even when starting far from the desired zero or minimum [1].

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Broyden, C.G. 1965, *Mathematics of Computation*, vol. 19, pp. 577–593. [2]

Chapter 10. Minimization or Maximization of Functions

10.0 Introduction

In a nutshell: You are given a single function f that depends on one or more independent variables. You want to find the value of those variables where f takes on a maximum or a minimum value. You can then calculate what value of f is achieved at the maximum or minimum. The tasks of maximization and minimization are trivially related to each other, since one person's function f could just as well be another's $-f$. The computational desiderata are the usual ones: Do it quickly, cheaply, and in small memory. Often the computational effort is dominated by the cost of evaluating f (and also perhaps its partial derivatives with respect to all variables, if the chosen algorithm requires them). In such cases the desiderata are sometimes replaced by the simple surrogate: Evaluate f as few times as possible.

An extremum (maximum or minimum point) can be either *global* (truly the highest or lowest function value) or *local* (the highest or lowest in a finite neighborhood and not on the boundary of that neighborhood). (See Figure 10.0.1.) Finding a global extremum is, in general, a very difficult problem. Two standard heuristics are widely used: (i) find local extrema starting from widely varying starting values of the independent variables (perhaps chosen quasi-randomly, as in §7.7), and then pick the most extreme of these (if they are not all the same); or (ii) perturb a local extremum by taking a finite amplitude step away from it, and then see if your routine returns you to a better point, or "always" to the same one. Relatively recently, so-called "simulated annealing methods" (§10.9) have demonstrated important successes on a variety of global extremization problems.

Our chapter title could just as well be *optimization*, which is the usual name for this very large field of numerical research. The importance ascribed to the various tasks in this field depends strongly on the particular interests of whom you talk to. Economists, and some engineers, are particularly concerned with *constrained optimization*, where there are *a priori* limitations on the allowed values of independent variables. For example, the production of wheat in the U.S. must be a nonnegative number. One particularly well-developed area of constrained optimization is *linear programming*, where both the function to be optimized and the constraints happen to be linear functions of the independent variables. Section 10.8, which is otherwise somewhat disconnected from the rest of the material that we have chosen to include in this chapter, implements the so-called "simplex algorithm" for linear programming problems.

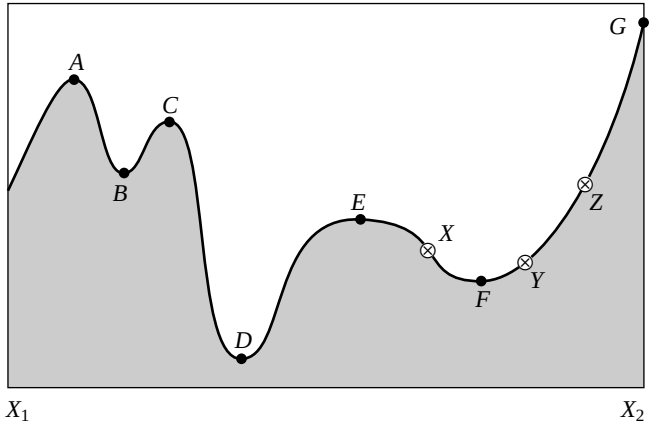


Figure 10.0.1. Extrema of a function in an interval. Points A, C, and E are local, but not global maxima. Points B and F are local, but not global minima. The global maximum occurs at G, which is on the boundary of the interval so that the derivative of the function need not vanish there. The global minimum is at D. At point E, derivatives higher than the first vanish, a situation which can cause difficulty for some algorithms. The points X, Y, and Z are said to "bracket" the minimum F, since Y is less than both X and Z.

One other section, §10.9, also lies outside of our main thrust, but for a different reason: so-called "annealing methods" are relatively new, so we do not yet know where they will ultimately fit into the scheme of things. However, these methods have solved some problems previously thought to be practically insoluble; they address directly the problem of finding global extrema in the presence of large numbers of undesired local extrema.

The other sections in this chapter constitute a selection of the best established algorithms in unconstrained minimization. (For definiteness, we will henceforth regard the optimization problem as that of minimization.) These sections are connected, with later ones depending on earlier ones. If you are just looking for the one "perfect" algorithm to solve your particular application, you may feel that we are telling you more than you want to know. Unfortunately, there is *no* perfect optimization algorithm. This is a case where we strongly urge you to try more than one method in comparative fashion. Your initial choice of method can be based on the following considerations:

- You must choose between methods that need only evaluations of the function to be minimized and methods that also require evaluations of the derivative of that function. In the multidimensional case, this derivative is the gradient, a vector quantity. Algorithms using the derivative are somewhat more powerful than those using only the function, but not always enough so as to compensate for the additional calculations of derivatives. We can easily construct examples favoring one approach or favoring the other. However, if you *can* compute derivatives, be prepared to try using them.
- For one-dimensional minimization (minimize a function of one variable) *without* calculation of the derivative, bracket the minimum as described in §10.1, and then use *Brent's method* as described in §10.2. If your function has a discontinuous second (or lower) derivative, then the parabolic

interpolations of Brent's method are of no advantage, and you might wish to use the simplest form of *golden section search*, as described in §10.1.
- For one-dimensional minimization *with* calculation of the derivative, §10.3 supplies a variant of Brent's method which makes limited use of the first derivative information. We shy away from the alternative of using derivative information to construct high-order interpolating polynomials. In our experience the improvement in convergence very near a smooth, analytic minimum does not make up for the tendency of polynomials sometimes to give wildly wrong interpolations at early stages, especially for functions that may have sharp, "exponential" features.

We now turn to the multidimensional case, both with and without computation of first derivatives.

- You must choose between methods that require storage of order N^2 and those that require only of order N, where N is the number of dimensions. For moderate values of N and reasonable memory sizes this is not a serious constraint. There will be, however, the occasional application where storage may be critical.
- We give in §10.4 a sometimes overlooked *downhill simplex method* due to Nelder and Mead. (This use of the word "simplex" is not to be confused with the simplex method of linear programming.) This method just crawls downhill in a straightforward fashion that makes almost no special assumptions about your function. This can be extremely slow, but it can also, in some cases, be extremely robust. Not to be overlooked is the fact that the code is concise and completely self-contained: a general N-dimensional minimization program in under 100 program lines! This method is most useful when the minimization calculation is only an incidental part of your overall problem. The storage requirement is of order N^2, and derivative calculations are not required.
- Section 10.5 deals with *direction-set methods*, of which *Powell's method* is the prototype. These are the methods of choice when you cannot easily calculate derivatives, and are not necessarily to be sneered at even if you can. Although derivatives are not needed, the method does require a one-dimensional minimization sub-algorithm such as Brent's method (see above). Storage is of order N^2.

There are two major families of algorithms for multidimensional minimization *with* calculation of first derivatives. Both families require a one-dimensional minimization sub-algorithm, which can itself either use, or not use, the derivative information, as you see fit (depending on the relative effort of computing the function and of its gradient vector). We do not think that either family dominates the other in all applications; you should think of them as available alternatives:

- The first family goes under the name *conjugate gradient methods*, as typified by the *Fletcher-Reeves algorithm* and the closely related and probably superior *Polak-Ribiere algorithm*. Conjugate gradient methods require only of order a few times N storage, require derivative calculations and

one-dimensional sub-minimization. Turn to §10.6 for detailed discussion and implementation.

- The second family goes under the names *quasi-Newton* or *variable metric* methods, as typified by the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to just as *Fletcher-Powell*) or the closely related *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* algorithm. These methods require of order N^2 storage, require derivative calculations and one-dimensional sub-minimization. Details are in §10.7.

You are now ready to proceed with scaling the peaks (and/or plumbing the depths) of practical optimization.

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press).

Gill, P.E., Murray, W., and Wright, M.H. 1981, *Practical Optimization* (New York: Academic Press).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 17.

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall).

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

10.1 Golden Section Search in One Dimension

Recall how the bisection method finds roots of functions in one dimension (§9.1): The root is supposed to have been bracketed in an interval (a, b). One then evaluates the function at an intermediate point x and obtains a new, smaller bracketing interval, either (a, x) or (x, b). The process continues until the bracketing interval is acceptably small. It is optimal to choose x to be the midpoint of (a, b) so that the decrease in the interval length is maximized when the function is as uncooperative as it can be, i.e., when the luck of the draw forces you to take the bigger bisected segment.

There is a precise, though slightly subtle, translation of these considerations to the minimization problem: What does it mean to *bracket* a minimum? A root of a function is known to be bracketed by a pair of points, a and b, when the function has opposite sign at those two points. A minimum, by contrast, is known to be bracketed only when there is a *triplet* of points, $a < b < c$ (or $c < b < a$), such that $f(b)$ is less than both $f(a)$ and $f(c)$. In this case we know that the function (if it is nonsingular) has a minimum in the interval (a, c).

The analog of bisection is to choose a new point x, either between a and b or between b and c. Suppose, to be specific, that we make the latter choice. Then we evaluate $f(x)$. If $f(b) < f(x)$, then the new bracketing triplet of points is (a, b, x);

one-dimensional sub-minimization. Turn to §10.6 for detailed discussion and implementation.

- The second family goes under the names *quasi-Newton* or *variable metric* methods, as typified by the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to just as *Fletcher-Powell*) or the closely related *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* algorithm. These methods require of order N^2 storage, require derivative calculations and one-dimensional sub-minimization. Details are in §10.7.

You are now ready to proceed with scaling the peaks (and/or plumbing the depths) of practical optimization.

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press).

Gill, P.E., Murray, W., and Wright, M.H. 1981, *Practical Optimization* (New York: Academic Press).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 17.

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall).

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

10.1 Golden Section Search in One Dimension

Recall how the bisection method finds roots of functions in one dimension (§9.1): The root is supposed to have been bracketed in an interval (a, b). One then evaluates the function at an intermediate point x and obtains a new, smaller bracketing interval, either (a, x) or (x, b). The process continues until the bracketing interval is acceptably small. It is optimal to choose x to be the midpoint of (a, b) so that the decrease in the interval length is maximized when the function is as uncooperative as it can be, i.e., when the luck of the draw forces you to take the bigger bisected segment.

There is a precise, though slightly subtle, translation of these considerations to the minimization problem: What does it mean to *bracket* a minimum? A root of a function is known to be bracketed by a pair of points, a and b, when the function has opposite sign at those two points. A minimum, by contrast, is known to be bracketed only when there is a *triplet* of points, $a < b < c$ (or $c < b < a$), such that $f(b)$ is less than both $f(a)$ and $f(c)$. In this case we know that the function (if it is nonsingular) has a minimum in the interval (a, c).

The analog of bisection is to choose a new point x, either between a and b or between b and c. Suppose, to be specific, that we make the latter choice. Then we evaluate $f(x)$. If $f(b) < f(x)$, then the new bracketing triplet of points is (a, b, x);

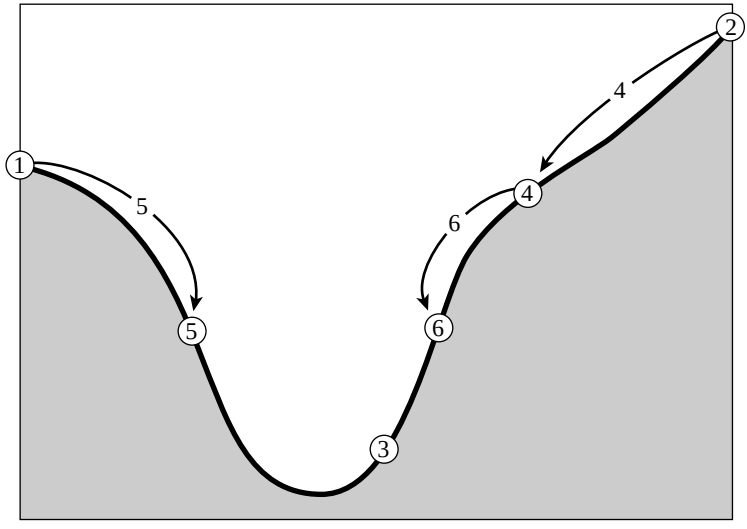


Figure 10.1.1. Successive bracketing of a minimum. The minimum is originally bracketed by points 1,3,2. The function is evaluated at 4, which replaces 2; then at 5, which replaces 1; then at 6, which replaces 4. The rule at each stage is to keep a center point that is lower than the two outside points. After the steps shown, the minimum is bracketed by points 5,3,6.

contrariwise, if $f(b) > f(x)$, then the new bracketing triplet is (b, x, c). In all cases the middle point of the new triplet is the abscissa whose ordinate is the best minimum achieved so far; see Figure 10.1.1. We continue the process of bracketing until the distance between the two outer points of the triplet is tolerably small.

How small is "tolerably" small? For a minimum located at a value b, you might naively think that you will be able to bracket it in as small a range as $(1 - \epsilon)b < b < (1 + \epsilon)b$, where ϵ is your computer's floating-point precision, a number like 3×10^{-8} (single precision) or 10^{-15} (double precision). Not so! In general, the shape of your function $f(x)$ near b will be given by Taylor's theorem

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x - b)^2 \tag{10.1.1}$$

The second term will be negligible compared to the first (that is, will be a factor ϵ smaller and will act just like zero when added to it) whenever

$$|x - b| < \sqrt{\epsilon}|b| \sqrt{\frac{2\,|f(b)|}{b^2 f''(b)}} \tag{10.1.2}$$

The reason for writing the right-hand side in this way is that, for most functions, the final square root is a number of order unity. Therefore, as a rule of thumb, it is hopeless to ask for a bracketing interval of width less than $\sqrt{\epsilon}$ times its central value, a fractional width of only about 10^{-4} (single precision) or 3×10^{-8} (double precision). Knowing this inescapable fact will save you a lot of useless bisections!

The minimum-finding routines of this chapter will often call for a user-supplied argument `tol`, and return with an abscissa whose fractional precision is about $\pm$`tol` (bracketing interval of fractional size about $2\times$`tol`). Unless you have a better

estimate for the right-hand side of equation (10.1.2), you should set `tol` equal to (not much less than) the square root of your machine's floating-point precision, since smaller values will gain you nothing.

It remains to decide on a strategy for choosing the new point x, given (a, b, c). Suppose that b is a fraction w of the way between a and c, i.e.

$$\frac{b-a}{c-a} = w \qquad \frac{c-b}{c-a} = 1-w \tag{10.1.3}$$

Also suppose that our next trial point x is an additional fraction z beyond b,

$$\frac{x-b}{c-a} = z \tag{10.1.4}$$

Then the next bracketing segment will either be of length $w+z$ relative to the current one, or else of length $1-w$. If we want to minimize the worst case possibility, then we will choose z to make these equal, namely

$$z = 1-2w \tag{10.1.5}$$

We see at once that the new point is the symmetric point to b in the original interval, namely with $|b-a|$ equal to $|x-c|$. This implies that the point x lies in the larger of the two segments (z is positive only if $w < 1/2$).

But where in the larger segment? Where did the value of w itself come from? Presumably from the previous stage of applying our same strategy. Therefore, if z is chosen to be optimal, then so was w before it. This *scale similarity* implies that x should be the same fraction of the way from b to c (if that is the bigger segment) as was b from a to c, in other words,

$$\frac{z}{1-w} = w \tag{10.1.6}$$

Equations (10.1.5) and (10.1.6) give the quadratic equation

$$w^2 - 3w + 1 = 0 \qquad \text{yielding} \qquad w = \frac{3-\sqrt{5}}{2} \approx 0.38197 \tag{10.1.7}$$

In other words, the optimal bracketing interval (a, b, c) has its middle point b a fractional distance 0.38197 from one end (say, a), and 0.61803 from the other end (say, b). These fractions are those of the so-called *golden mean* or *golden section*, whose supposedly aesthetic properties hark back to the ancient Pythagoreans. This optimal method of function minimization, the analog of the bisection method for finding zeros, is thus called the *golden section search*, summarized as follows:

Given, at each stage, a bracketing triplet of points, the next point to be tried is that which is a fraction 0.38197 into the larger of the two intervals (measuring from the central point of the triplet). If you start out with a bracketing triplet whose segments are not in the golden ratios, the procedure of choosing successive points at the golden mean point of the larger segment will quickly converge you to the proper, self-replicating ratios.

The golden section search guarantees that each new function evaluation will (after self-replicating ratios have been achieved) bracket the minimum to an interval

just 0.61803 times the size of the preceding interval. This is comparable to, but not quite as good as, the 0.50000 that holds when finding roots by bisection. Note that the convergence is *linear* (in the language of Chapter 9), meaning that successive significant figures are won linearly with additional function evaluations. In the next section we will give a superlinear method, where the rate at which successive significant figures are liberated increases with each successive function evaluation.

Routine for Initially Bracketing a Minimum

The preceding discussion has assumed that you are able to bracket the minimum in the first place. We consider this initial bracketing to be an essential part of any one-dimensional minimization. There are some one-dimensional algorithms that do not require a rigorous initial bracketing. However, we would *never* trade the secure feeling of *knowing* that a minimum is "in there somewhere" for the dubious reduction of function evaluations that these nonbracketing routines may promise. Please bracket your minima (or, for that matter, your zeros) before isolating them!

There is not much theory as to how to do this bracketing. Obviously you want to step downhill. But how far? We like to take larger and larger steps, starting with some (wild?) initial guess and then increasing the stepsize at each step either by a constant factor, or else by the result of a parabolic extrapolation of the preceding points that is designed to take us to the extrapolated turning point. It doesn't much matter if the steps get big. After all, we are stepping downhill, so we already have the left and middle points of the bracketing triplet. We just need to take a big enough step to stop the downhill trend and get a high third point.

Our standard routine is this:

```
      SUBROUTINE mnbrak(ax,bx,cx,fa,fb,fc,func)
      REAL ax,bx,cx,fa,fb,fc,func,GOLD,GLIMIT,TINY
      EXTERNAL func
      PARAMETER (GOLD=1.618034, GLIMIT=100., TINY=1.e-20)
        Given a function func, and given distinct initial points ax and bx, this routine searches
        in the downhill direction (defined by the function as evaluated at the initial points) and
        returns new points ax, bx, cx that bracket a minimum of the function. Also returned are
        the function values at the three points, fa, fb, and fc.
        Parameters: GOLD is the default ratio by which successive intervals are magnified; GLIMIT
        is the maximum magnification allowed for a parabolic-fit step.
      REAL dum,fu,q,r,u,ulim
      fa=func(ax)
      fb=func(bx)
      if(fb.gt.fa)then                    Switch roles of a and b so that we can go downhill in the
        dum=ax                              direction from a to b.
        ax=bx
        bx=dum
        dum=fb
        fb=fa
        fa=dum
      endif
      cx=bx+GOLD*(bx-ax)                  First guess for c.
      fc=func(cx)
1     if(fb.ge.fc)then                    "do while": keep returning here until we bracket.
        r=(bx-ax)*(fb-fc)                 Compute u by parabolic extrapolation from a, b, c. TINY
        q=(bx-cx)*(fb-fa)                   is used to prevent any possible division by zero.
        u=bx-((bx-cx)*q-(bx-ax)*r)/(2.*sign(max(abs(q-r),TINY),q-r))
        ulim=bx+GLIMIT*(cx-bx)            We won't go farther than this. Test various possibilities:
        if((bx-u)*(u-cx).gt.0.)then         Parabolic u is between b and c: try it.
          fu=func(u)
```

```
        if(fu.lt.fc)then          Got a minimum between b and c.
            ax=bx
            fa=fb
            bx=u
            fb=fu
            return
        else if(fu.gt.fb)then     Got a minimum between between a and u.
            cx=u
            fc=fu
            return
        endif
        u=cx+GOLD*(cx-bx)         Parabolic fit was no use. Use default magnification.
        fu=func(u)
    else if((cx-u)*(u-ulim).gt.0.)then        Parabolic fit is between c and its allowed
        fu=func(u)                                limit.
        if(fu.lt.fc)then
            bx=cx
            cx=u
            u=cx+GOLD*(cx-bx)
            fb=fc
            fc=fu
            fu=func(u)
        endif
    else if((u-ulim)*(ulim-cx).ge.0.)then     Limit parabolic u to maximum allowed
        u=ulim                                    value.
        fu=func(u)
    else                          Reject parabolic u, use default magnification.
        u=cx+GOLD*(cx-bx)
        fu=func(u)
    endif
    ax=bx                         Eliminate oldest point and continue.
    bx=cx
    cx=u
    fa=fb
    fb=fc
    fc=fu
    goto 1
endif
return
END
```

(Because of the housekeeping involved in moving around three or four points and their function values, the above program ends up looking deceptively formidable. That is true of several other programs in this chapter as well. The underlying ideas, however, are quite simple.)

Routine for Golden Section Search

```
FUNCTION golden(ax,bx,cx,f,tol,xmin)
REAL golden,ax,bx,cx,tol,xmin,f,R,C
EXTERNAL f
PARAMETER (R=.61803399,C=1.-R)
    Given a function f, and given a bracketing triplet of abscissas ax, bx, cx (such that bx is
    between ax and cx, and f(bx) is less than both f(ax) and f(cx)), this routine performs
    a golden section search for the minimum, isolating it to a fractional precision of about
    tol. The abscissa of the minimum is returned as xmin, and the minimum function value
    is returned as golden, the returned function value.
    Parameters: The golden ratios.
REAL f1,f2,x0,x1,x2,x3
x0=ax                             At any given time we will keep track of four points, x0,x1,x2,x3.
```

```
      x3=cx
      if(abs(cx-bx).gt.abs(bx-ax))then       Make x0 to x1 the smaller segment,
          x1=bx
          x2=bx+C*(cx-bx)                    and fill in the new point to be tried.
      else
          x2=bx
          x1=bx-C*(bx-ax)
      endif
      f1=f(x1)                               The initial function evaluations. Note that we never need to
      f2=f(x2)                                   evaluate the function at the original endpoints.
1     if(abs(x3-x0).gt.tol*(abs(x1)+abs(x2)))then  Do-while loop: we keep returning here.
          if(f2.lt.f1)then                   One possible outcome,
              x0=x1                          its housekeeping,
              x1=x2
              x2=R*x1+C*x3
              f1=f2
              f2=f(x2)                       and a new function evaluation.
          else                               The other outcome,
              x3=x2
              x2=x1
              x1=R*x2+C*x0
              f2=f1
              f1=f(x1)                       and its new function evaluation.
          endif
      goto 1                                 Back to see if we are done.
      endif
      if(f1.lt.f2)then                       We are done. Output the best of the two current values.
          golden=f1
          xmin=x1
      else
          golden=f2
          xmin=x2
      endif
      return
      END
```

10.2 Parabolic Interpolation and Brent's Method in One Dimension

We already tipped our hand about the desirability of parabolic interpolation in the previous section's `mnbrak` routine, but it is now time to be more explicit. A golden section search is designed to handle, in effect, the worst possible case of function minimization, with the uncooperative minimum hunted down and cornered like a scared rabbit. But why assume the worst? If the function is nicely parabolic near to the minimum — surely the generic case for sufficiently smooth functions — then the parabola fitted through any three points ought to take us in a single leap to the minimum, or at least very near to it (see Figure 10.2.1). Since we want to find an abscissa rather than an ordinate, the procedure is technically called *inverse parabolic interpolation*.

The formula for the abscissa x that is the minimum of a parabola through three points $f(a)$, $f(b)$, and $f(c)$ is

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \tag{10.2.1}$$

```
      x3=cx
      if(abs(cx-bx).gt.abs(bx-ax))then       Make x0 to x1 the smaller segment,
          x1=bx
          x2=bx+C*(cx-bx)                    and fill in the new point to be tried.
      else
          x2=bx
          x1=bx-C*(bx-ax)
      endif
      f1=f(x1)                               The initial function evaluations. Note that we never need to
      f2=f(x2)                                   evaluate the function at the original endpoints.
1     if(abs(x3-x0).gt.tol*(abs(x1)+abs(x2)))then  Do-while loop: we keep returning here.
          if(f2.lt.f1)then                   One possible outcome,
              x0=x1                          its housekeeping,
              x1=x2
              x2=R*x1+C*x3
              f1=f2
              f2=f(x2)                       and a new function evaluation.
          else                               The other outcome,
              x3=x2
              x2=x1
              x1=R*x2+C*x0
              f2=f1
              f1=f(x1)                       and its new function evaluation.
          endif
      goto 1                                 Back to see if we are done.
      endif
      if(f1.lt.f2)then                       We are done. Output the best of the two current values.
          golden=f1
          xmin=x1
      else
          golden=f2
          xmin=x2
      endif
      return
      END
```

10.2 Parabolic Interpolation and Brent's Method in One Dimension

We already tipped our hand about the desirability of parabolic interpolation in the previous section's `mnbrak` routine, but it is now time to be more explicit. A golden section search is designed to handle, in effect, the worst possible case of function minimization, with the uncooperative minimum hunted down and cornered like a scared rabbit. But why assume the worst? If the function is nicely parabolic near to the minimum — surely the generic case for sufficiently smooth functions — then the parabola fitted through any three points ought to take us in a single leap to the minimum, or at least very near to it (see Figure 10.2.1). Since we want to find an abscissa rather than an ordinate, the procedure is technically called *inverse parabolic interpolation*.

The formula for the abscissa x that is the minimum of a parabola through three points $f(a)$, $f(b)$, and $f(c)$ is

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \tag{10.2.1}$$

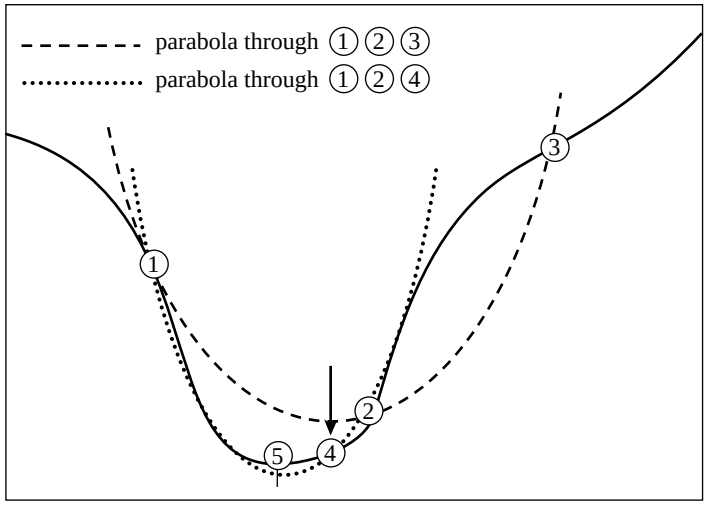


Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn through points 1,4,2. The minimum of this parabola is at 5, which is close to the minimum of the function.

as you can easily derive. This formula fails only if the three points are collinear, in which case the denominator is zero (minimum of the parabola is infinitely far away). Note, however, that (10.2.1) is as happy jumping to a parabolic maximum as to a minimum. No minimization scheme that depends solely on (10.2.1) is likely to succeed in practice.

The exacting task is to invent a scheme that relies on a sure-but-slow technique, like golden section search, when the function is not cooperative, but that switches over to (10.2.1) when the function allows. The task is nontrivial for several reasons, including these: (i) The housekeeping needed to avoid unnecessary function evaluations in switching between the two methods can be complicated. (ii) Careful attention must be given to the "endgame," where the function is being evaluated very near to the roundoff limit of equation (10.1.2). (iii) The scheme for detecting a cooperative versus noncooperative function must be very robust.

Brent's method [1] is up to the task in all particulars. At any particular stage, it is keeping track of six function points (not necessarily all distinct), a, b, u, v, w and x, defined as follows: the minimum is bracketed between a and b; x is the point with the very least function value found so far (or the most recent one in case of a tie); w is the point with the second least function value; v is the previous value of w; u is the point at which the function was evaluated most recently. Also appearing in the algorithm is the point x_m, the midpoint between a and b; however, the function is not evaluated there.

You can read the code below to understand the method's logical organization. Mention of a few general principles here may, however, be helpful: Parabolic interpolation is attempted, fitting through the points x, v, and w. To be acceptable, the parabolic step must (i) fall within the bounding interval (a, b), and (ii) imply a movement from the best current value x that is *less* than half the movement of the *step before last*. This second criterion insures that the parabolic steps are actually

converging to something, rather than, say, bouncing around in some nonconvergent limit cycle. In the worst possible case, where the parabolic steps are acceptable but useless, the method will approximately alternate between parabolic steps and golden sections, converging in due course by virtue of the latter. The reason for comparing to the step *before* last seems essentially heuristic: Experience shows that it is better not to "punish" the algorithm for a single bad step if it can make it up on the next one.

Another principle exemplified in the code is never to evaluate the function less than a distance `tol` from a point already evaluated (or from a known bracketing point). The reason is that, as we saw in equation (10.1.2), there is simply no information content in doing so: the function will differ from the value already evaluated only by an amount of order the roundoff error. Therefore in the code below you will find several tests and modifications of a potential new point, imposing this restriction. This restriction also interacts subtly with the test for "doneness," which the method takes into account.

A typical ending configuration for Brent's method is that a and b are $2 \times x \times$ `tol` apart, with x (the best abscissa) at the midpoint of a and b, and therefore fractionally accurate to $\pm$`tol`.

Indulge us a final reminder that `tol` should generally be no smaller than the square root of your machine's floating-point precision.

```
FUNCTION brent(ax,bx,cx,f,tol,xmin)
INTEGER ITMAX
REAL brent,ax,bx,cx,tol,xmin,f,CGOLD,ZEPS
EXTERNAL f
PARAMETER (ITMAX=100,CGOLD=.3819660,ZEPS=1.0e-10)
    Given a function f, and given a bracketing triplet of abscissas ax, bx, cx (such that bx is
    between ax and cx, and f(bx) is less than both f(ax) and f(cx)), this routine isolates
    the minimum to a fractional precision of about tol using Brent's method. The abscissa of
    the minimum is returned as xmin, and the minimum function value is returned as brent,
    the returned function value.
    Parameters: Maximum allowed number of iterations; golden ratio; and a small number that
    protects against trying to achieve fractional accuracy for a minimum that happens to be
    exactly zero.
INTEGER iter
REAL a,b,d,e,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm
a=min(ax,cx)                        a and b must be in ascending order, though the input
b=max(ax,cx)                            abscissas need not be.
v=bx                                Initializations...
w=v
x=v
e=0.                                This will be the distance moved on the step before last.
fx=f(x)
fv=fx
fw=fx
do 11 iter=1,ITMAX                  Main program loop.
    xm=0.5*(a+b)
    tol1=tol*abs(x)+ZEPS
    tol2=2.*tol1
    if(abs(x-xm).le.(tol2-.5*(b-a))) goto 3      Test for done here.
    if(abs(e).gt.tol1) then                      Construct a trial parabolic fit.
        r=(x-w)*(fx-fv)
        q=(x-v)*(fx-fw)
        p=(x-v)*q-(x-w)*r
        q=2.*(q-r)
        if(q.gt.0.) p=-p
        q=abs(q)
        etemp=e
        e=d
```

```
            if(abs(p).ge.abs(.5*q*etemp).or.p.le.q*(a-x).or.
*                p.ge.q*(b-x)) goto 1
              The above conditions determine the acceptability of the parabolic fit. Here it is o.k.:
            d=p/q                       Take the parabolic step.
            u=x+d
            if(u-a.lt.tol2 .or. b-u.lt.tol2) d=sign(tol1,xm-x)
            goto 2                      Skip over the golden section step.
        endif
1       if(x.ge.xm) then                We arrive here for a golden section step, which we take
            e=a-x                           into the larger of the two segments.
        else
            e=b-x
        endif
        d=CGOLD*e                       Take the golden section step.
2       if(abs(d).ge.tol1) then         Arrive here with d computed either from parabolic fit, or
            u=x+d                           else from golden section.
        else
            u=x+sign(tol1,d)
        endif
        fu=f(u)                         This is the one function evaluation per iteration,
        if(fu.le.fx) then               and now we have to decide what to do with our function
            if(u.ge.x) then                 evaluation. Housekeeping follows:
                a=x
            else
                b=x
            endif
            v=w
            fv=fw
            w=x
            fw=fx
            x=u
            fx=fu
        else
            if(u.lt.x) then
                a=u
            else
                b=u
            endif
            if(fu.le.fw .or. w.eq.x) then
                v=w
                fv=fw
                w=u
                fw=fu
            else if(fu.le.fv .or. v.eq.x .or. v.eq.w) then
                v=u
                fv=fu
            endif
        endif                           Done with housekeeping. Back for another iteration.
    enddo 11
    pause 'brent exceed maximum iterations'
3   xmin=x                              Arrive here ready to exit with best values.
    brent=fx
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §8.2.

10.3 One-Dimensional Search with First Derivatives

Here we want to accomplish precisely the same goal as in the previous section, namely to isolate a functional minimum that is bracketed by the triplet of abscissas (a, b, c), but utilizing an additional capability to compute the function's first derivative as well as its value.

In principle, we might simply search for a zero of the derivative, ignoring the function value information, using a root finder like `rtflsp` or `zbrent` (§§9.2–9.3). It doesn't take long to reject *that* idea: How do we distinguish maxima from minima? Where do we go from initial conditions where the derivatives on one or both of the outer bracketing points indicate that "downhill" is in the direction *out* of the bracketed interval?

We don't want to give up our strategy of maintaining a rigorous bracket on the minimum at all times. The only way to keep such a bracket is to update it using function (not derivative) information, with the central point in the bracketing triplet always that with the lowest function value. Therefore the role of the derivatives can only be to help us choose new trial points within the bracket.

One school of thought is to "use everything you've got": Compute a polynomial of relatively high order (cubic or above) that agrees with some number of previous function and derivative evaluations. For example, there is a unique cubic that agrees with function and derivative at two points, and one can jump to the interpolated minimum of that cubic (if there is a minimum within the bracket). Suggested by Davidon and others, formulas for this tactic are given in [1].

We like to be more conservative than this. Once superlinear convergence sets in, it hardly matters whether its order is moderately lower or higher. In practical problems that we have met, most function evaluations are spent in getting globally close enough to the minimum for superlinear convergence to commence. So we are more worried about all the funny "stiff" things that high-order polynomials can do (cf. Figure 3.0.1b), and about their sensitivities to roundoff error.

This leads us to use derivative information only as follows: The sign of the derivative at the central point of the bracketing triplet (a, b, c) indicates uniquely whether the next test point should be taken in the interval (a, b) or in the interval (b, c). The value of this derivative and of the derivative at the second-best-so-far point are extrapolated to zero by the secant method (inverse linear interpolation), which by itself is superlinear of order 1.618. (The golden mean again: see [1], p. 57.) We impose the same sort of restrictions on this new trial point as in Brent's method. If the trial point must be rejected, we *bisect* the interval under scrutiny.

Yes, we are fuddy-duddies when it comes to making flamboyant use of derivative information in one-dimensional minimization. But we have met too many functions whose computed "derivatives" *don't* integrate up to the function value and *don't* accurately point the way to the minimum, usually because of roundoff errors, sometimes because of truncation error in the method of derivative evaluation.

You will see that the following routine is closely modeled on `brent` in the previous section.

10.3 One-Dimensional Search with First Derivatives

Here we want to accomplish precisely the same goal as in the previous section, namely to isolate a functional minimum that is bracketed by the triplet of abscissas (a, b, c), but utilizing an additional capability to compute the function's first derivative as well as its value.

In principle, we might simply search for a zero of the derivative, ignoring the function value information, using a root finder like `rtflsp` or `zbrent` (§§9.2–9.3). It doesn't take long to reject *that* idea: How do we distinguish maxima from minima? Where do we go from initial conditions where the derivatives on one or both of the outer bracketing points indicate that "downhill" is in the direction *out* of the bracketed interval?

We don't want to give up our strategy of maintaining a rigorous bracket on the minimum at all times. The only way to keep such a bracket is to update it using function (not derivative) information, with the central point in the bracketing triplet always that with the lowest function value. Therefore the role of the derivatives can only be to help us choose new trial points within the bracket.

One school of thought is to "use everything you've got": Compute a polynomial of relatively high order (cubic or above) that agrees with some number of previous function and derivative evaluations. For example, there is a unique cubic that agrees with function and derivative at two points, and one can jump to the interpolated minimum of that cubic (if there is a minimum within the bracket). Suggested by Davidon and others, formulas for this tactic are given in [1].

We like to be more conservative than this. Once superlinear convergence sets in, it hardly matters whether its order is moderately lower or higher. In practical problems that we have met, most function evaluations are spent in getting globally close enough to the minimum for superlinear convergence to commence. So we are more worried about all the funny "stiff" things that high-order polynomials can do (cf. Figure 3.0.1b), and about their sensitivities to roundoff error.

This leads us to use derivative information only as follows: The sign of the derivative at the central point of the bracketing triplet (a, b, c) indicates uniquely whether the next test point should be taken in the interval (a, b) or in the interval (b, c). The value of this derivative and of the derivative at the second-best-so-far point are extrapolated to zero by the secant method (inverse linear interpolation), which by itself is superlinear of order 1.618. (The golden mean again: see [1], p. 57.) We impose the same sort of restrictions on this new trial point as in Brent's method. If the trial point must be rejected, we *bisect* the interval under scrutiny.

Yes, we are fuddy-duddies when it comes to making flamboyant use of derivative information in one-dimensional minimization. But we have met too many functions whose computed "derivatives" *don't* integrate up to the function value and *don't* accurately point the way to the minimum, usually because of roundoff errors, sometimes because of truncation error in the method of derivative evaluation.

You will see that the following routine is closely modeled on `brent` in the previous section.

```
      FUNCTION dbrent(ax,bx,cx,f,df,tol,xmin)
      INTEGER ITMAX
      REAL dbrent,ax,bx,cx,tol,xmin,df,f,ZEPS
      EXTERNAL df,f
      PARAMETER (ITMAX=100,ZEPS=1.0e-10)
```

Given a function f and its derivative function df, and given a bracketing triplet of abscissas ax, bx, cx [such that bx is between ax and cx, and f(bx) is less than both f(ax) and f(cx)], this routine isolates the minimum to a fractional precision of about tol using a modification of Brent's method that uses derivatives. The abscissa of the minimum is returned as xmin, and the minimum function value is returned as dbrent, the returned function value.

```
      INTEGER iter
      REAL a,b,d,d1,d2,du,dv,dw,dx,e,fu,fv,fw,fx,olde,tol1,tol2,
     *     u,u1,u2,v,w,x,xm
```

Comments following will point out only differences from the routine brent. Read that routine first.

```
      LOGICAL ok1,ok2
      a=min(ax,cx)
      b=max(ax,cx)
      v=bx
      w=v
      x=v
      e=0.
      fx=f(x)
      fv=fx
      fw=fx
      dx=df(x)
      dv=dx
      dw=dx
      do 11 iter=1,ITMAX
        xm=0.5*(a+b)
        tol1=tol*abs(x)+ZEPS
        tol2=2.*tol1
        if(abs(x-xm).le.(tol2-.5*(b-a))) goto 3
        if(abs(e).gt.tol1) then
          d1=2.*(b-a)
          d2=d1
          if(dw.ne.dx) d1=(w-x)*dx/(dx-dw)
          if(dv.ne.dx) d2=(v-x)*dx/(dx-dv)
          u1=x+d1
          u2=x+d2
          ok1=((a-u1)*(u1-b).gt.0.).and.(dx*d1.le.0.)
          ok2=((a-u2)*(u2-b).gt.0.).and.(dx*d2.le.0.)
          olde=e
          e=d
          if(.not.(ok1.or.ok2))then
            goto 1
          else if (ok1.and.ok2)then
            if(abs(d1).lt.abs(d2))then
              d=d1
            else
              d=d2
            endif
          else if (ok1)then
            d=d1
          else
            d=d2
          endif
          if(abs(d).gt.abs(0.5*olde))goto 1
          u=x+d
          if(u-a.lt.tol2 .or. b-u.lt.tol2) d=sign(tol1,xm-x)
          goto 2
```

Initialize these d's to an out-of-bracket value.

Secant method with one point.

And the other.

Which of these two estimates of d shall we take? We will insist that they be within the bracket, and on the side pointed to by the derivative at x:

Movement on the step before last.

Take only an acceptable d, and if both are acceptable, then take the smallest one.

```
        endif
1       if(dx.ge.0.) then          Decide which segment by the sign of the derivative.
            e=a-x
        else
            e=b-x
        endif
        d=0.5*e                    Bisect, not golden section.
2       if(abs(d).ge.tol1) then
            u=x+d
            fu=f(u)
        else
            u=x+sign(tol1,d)
            fu=f(u)
            if(fu.gt.fx)goto 3     If the minimum step in the downhill direction takes us uphill,
        endif                          then we are done.
        du=df(u)                   Now all the housekeeping, sigh.
        if(fu.le.fx) then
            if(u.ge.x) then
                a=x
            else
                b=x
            endif
            v=w
            fv=fw
            dv=dw
            w=x
            fw=fx
            dw=dx
            x=u
            fx=fu
            dx=du
        else
            if(u.lt.x) then
                a=u
            else
                b=u
            endif
            if(fu.le.fw .or. w.eq.x) then
                v=w
                fv=fw
                dv=dw
                w=u
                fw=fu
                dw=du
            else if(fu.le.fv .or. v.eq.x .or. v.eq.w) then
                v=u
                fv=fu
                dv=du
            endif
        endif
    enddo 11
    pause 'dbrent exceeded maximum iterations'
3   xmin=x
    dbrent=fx
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 55; 454–458. [1]

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), p. 78.

10.4 Downhill Simplex Method in Multidimensions

With this section we begin consideration of multidimensional minimization, that is, finding the minimum of a function of more than one independent variable. This section stands apart from those which follow, however: All of the algorithms after this section will make explicit use of a one-dimensional minimization algorithm as a part of their computational strategy. This section implements an entirely self-contained strategy, in which one-dimensional minimization does not figure.

The *downhill simplex method* is due to Nelder and Mead [1]. The method requires only function evaluations, not derivatives. It is not very efficient in terms of the number of function evaluations that it requires. Powell's method (§10.5) is almost surely faster in all likely applications. However, the downhill simplex method may frequently be the *best* method to use if the figure of merit is "get something working quickly" for a problem whose computational burden is small.

The method has a geometrical naturalness about it which makes it delightful to describe or work through:

A *simplex* is the geometrical figure consisting, in N dimensions, of $N+1$ points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron. (The *simplex method* of linear programming, described in §10.8, also makes use of the geometrical concept of a simplex. Otherwise it is completely unrelated to the algorithm that we are describing in this section.) In general we are only interested in simplexes that are nondegenerate, i.e., that enclose a finite inner N-dimensional volume. If any point of a nondegenerate simplex is taken as the origin, then the N other points define vector directions that span the N-dimensional vector space.

In one-dimensional minimization, it was possible to bracket a minimum, so that the success of a subsequent isolation was guaranteed. Alas! There is no analogous procedure in multidimensional space. For multidimensional minimization, the best we can do is give our algorithm a starting guess, that is, an N-vector of independent variables as the first point to try. The algorithm is then supposed to make its own way downhill through the unimaginable complexity of an N-dimensional topography, until it encounters a (local, at least) minimum.

The downhill simplex method must be started not just with a single point, but with $N+1$ points, defining an initial simplex. If you think of one of these points (it matters not which) as being your initial starting point $\mathbf{P}_0$, then you can take the other N points to be

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda \mathbf{e}_i \tag{10.4.1}$$

where the $\mathbf{e}_i$'s are N unit vectors, and where λ is a constant which is your guess of the problem's characteristic length scale. (Or, you could have different λ_i's for each vector direction.)

The downhill simplex method now takes a series of steps, most steps just moving the point of the simplex where the function is largest ("highest point") through the opposite face of the simplex to a lower point. These steps are called

10.4 Downhill Simplex Method in Multidimensions

With this section we begin consideration of multidimensional minimization, that is, finding the minimum of a function of more than one independent variable. This section stands apart from those which follow, however: All of the algorithms after this section will make explicit use of a one-dimensional minimization algorithm as a part of their computational strategy. This section implements an entirely self-contained strategy, in which one-dimensional minimization does not figure.

The *downhill simplex method* is due to Nelder and Mead [1]. The method requires only function evaluations, not derivatives. It is not very efficient in terms of the number of function evaluations that it requires. Powell's method (§10.5) is almost surely faster in all likely applications. However, the downhill simplex method may frequently be the *best* method to use if the figure of merit is "get something working quickly" for a problem whose computational burden is small.

The method has a geometrical naturalness about it which makes it delightful to describe or work through:

A *simplex* is the geometrical figure consisting, in N dimensions, of $N+1$ points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron. (The *simplex method* of linear programming, described in §10.8, also makes use of the geometrical concept of a simplex. Otherwise it is completely unrelated to the algorithm that we are describing in this section.) In general we are only interested in simplexes that are nondegenerate, i.e., that enclose a finite inner N-dimensional volume. If any point of a nondegenerate simplex is taken as the origin, then the N other points define vector directions that span the N-dimensional vector space.

In one-dimensional minimization, it was possible to bracket a minimum, so that the success of a subsequent isolation was guaranteed. Alas! There is no analogous procedure in multidimensional space. For multidimensional minimization, the best we can do is give our algorithm a starting guess, that is, an N-vector of independent variables as the first point to try. The algorithm is then supposed to make its own way downhill through the unimaginable complexity of an N-dimensional topography, until it encounters a (local, at least) minimum.

The downhill simplex method must be started not just with a single point, but with $N+1$ points, defining an initial simplex. If you think of one of these points (it matters not which) as being your initial starting point $\mathbf{P}_0$, then you can take the other N points to be

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda \mathbf{e}_i \tag{10.4.1}$$

where the $\mathbf{e}_i$'s are N unit vectors, and where λ is a constant which is your guess of the problem's characteristic length scale. (Or, you could have different λ_i's for each vector direction.)

The downhill simplex method now takes a series of steps, most steps just moving the point of the simplex where the function is largest ("highest point") through the opposite face of the simplex to a lower point. These steps are called

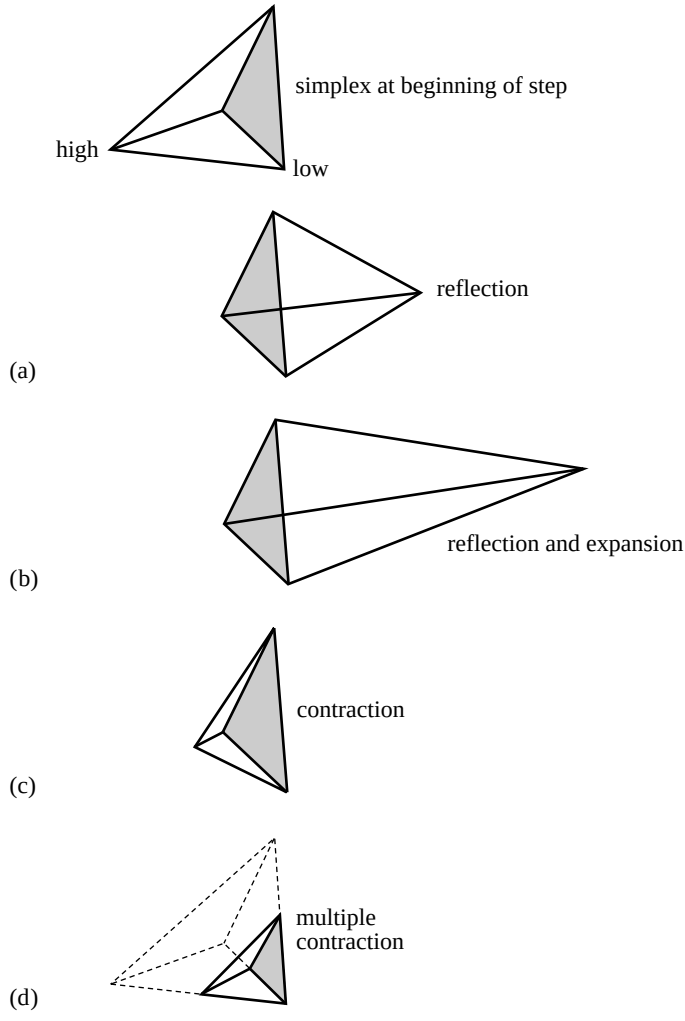


Figure 10.4.1. Possible outcomes for a step in the downhill simplex method. The simplex at the beginning of the step, here a tetrahedron, is shown, top. The simplex at the end of the step can be any one of (a) a reflection away from the high point, (b) a reflection and expansion away from the high point, (c) a contraction along one dimension from the high point, or (d) a contraction along all dimensions towards the low point. An appropriate sequence of such steps will always converge to a minimum of the function.

reflections, and they are constructed to conserve the volume of the simplex (hence maintain its nondegeneracy). When it can do so, the method expands the simplex in one or another direction to take larger steps. When it reaches a "valley floor," the method contracts itself in the transverse direction and tries to ooze down the valley. If there is a situation where the simplex is trying to "pass through the eye of a needle," it contracts itself in all directions, pulling itself in around its lowest (best) point. The routine name `amoeba` is intended to be descriptive of this kind of behavior; the basic moves are summarized in Figure 10.4.1.

Termination criteria can be delicate in any multidimensional minimization routine. Without bracketing, and with more than one independent variable, we no longer have the option of requiring a certain tolerance for a single independent

variable. We typically can identify one "cycle" or "step" of our multidimensional algorithm. It is then possible to terminate when the vector distance moved in that step is fractionally smaller in magnitude than some tolerance `tol`. Alternatively, we could require that the decrease in the function value in the terminating step be fractionally smaller than some tolerance `ftol`. Note that while `tol` should not usually be smaller than the square root of the machine precision, it is perfectly appropriate to let `ftol` be of order the machine precision (or perhaps slightly larger so as not to be diddled by roundoff).

Note well that either of the above criteria might be fooled by a single anomalous step that, for one reason or another, failed to get anywhere. Therefore, it is frequently a good idea to *restart* a multidimensional minimization routine at a point where it claims to have found a minimum. For this restart, you should reinitialize any ancillary input quantities. In the downhill simplex method, for example, you should reinitialize N of the $N + 1$ vertices of the simplex again by equation (10.4.1), with $\mathbf{P}_0$ being one of the vertices of the claimed minimum.

Restarts should never be very expensive; your algorithm did, after all, converge to the restart point once, and now you are starting the algorithm already there.

Consider, then, our N-dimensional amoeba:

```
      SUBROUTINE amoeba(p,y,mp,np,ndim,ftol,funk,iter)
      INTEGER iter,mp,ndim,np,NMAX,ITMAX
      REAL ftol,p(mp,np),y(mp),funk,TINY
      PARAMETER (NMAX=20,ITMAX=5000,TINY=1.e-10)    Maximum allowed dimensions and func-
      EXTERNAL funk                                   tion evaluations, and a small num-
C     USES amotry,funk                                ber.
        Multidimensional minimization of the function funk(x) where x(1:ndim) is a vector
        in ndim dimensions, by the downhill simplex method of Nelder and Mead. The matrix
        p(1:ndim+1,1:ndim) is input. Its ndim+1 rows are ndim-dimensional vectors which are
        the vertices of the starting simplex. Also input is the vector y(1:ndim+1), whose compo-
        nents must be pre-initialized to the values of funk evaluated at the ndim+1 vertices (rows)
        of p; and ftol the fractional convergence tolerance to be achieved in the function value
        (n.b.!). On output, p and y will have been reset to ndim+1 new points all within ftol of
        a minimum function value, and iter gives the number of function evaluations taken.
      INTEGER i,ihi,ilo,inhi,j,m,n
      REAL rtol,sum,swap,ysave,ytry,psum(NMAX),amotry
      iter=0
1     do 12 n=1,ndim                    Enter here when starting or have just overall contracted.
        sum=0.                          Recompute psum.
        do 11 m=1,ndim+1
          sum=sum+p(m,n)
        enddo 11
        psum(n)=sum
      enddo 12
2     ilo=1                             Enter here when have just changed a single point.
      if (y(1).gt.y(2)) then            Determine which point is the highest (worst), next-highest,
        ihi=1                             and lowest (best),
        inhi=2
      else
        ihi=2
        inhi=1
      endif
      do 13 i=1,ndim+1                  by looping over the points in the simplex.
        if(y(i).le.y(ilo)) ilo=i
        if(y(i).gt.y(ihi)) then
          inhi=ihi
          ihi=i
        else if(y(i).gt.y(inhi)) then
          if(i.ne.ihi) inhi=i
```

```
        endif
      enddo 13
      rtol=2.*abs(y(ihi)-y(ilo))/(abs(y(ihi))+abs(y(ilo))+TINY)
        Compute the fractional range from highest to lowest and return if satisfactory.
      if (rtol.lt.ftol) then        If returning, put best point and value in slot 1.
        swap=y(1)
        y(1)=y(ilo)
        y(ilo)=swap
        do 14 n=1,ndim
          swap=p(1,n)
          p(1,n)=p(ilo,n)
          p(ilo,n)=swap
        enddo 14
        return
      endif
      if (iter.ge.ITMAX) pause 'ITMAX exceeded in amoeba'
      iter=iter+2
        Begin a new iteration. First extrapolate by a factor −1 through the face of the simplex across
        from the high point, i.e., reflect the simplex from the high point.
      ytry=amotry(p,y,psum,mp,np,ndim,funk,ihi,-1.0)
      if (ytry.le.y(ilo)) then
        Gives a result better than the best point, so try an additional extrapolation by a factor 2.
        ytry=amotry(p,y,psum,mp,np,ndim,funk,ihi,2.0)
      else if (ytry.ge.y(inhi)) then
        The reflected point is worse than the second-highest, so look for an intermediate lower point,
        i.e., do a one-dimensional contraction.
        ysave=y(ihi)
        ytry=amotry(p,y,psum,mp,np,ndim,funk,ihi,0.5)
        if (ytry.ge.ysave) then     Can't seem to get rid of that high point. Better contract
          do 16 i=1,ndim+1              around the lowest (best) point.
            if(i.ne.ilo)then
              do 15 j=1,ndim
                psum(j)=0.5*(p(i,j)+p(ilo,j))
                p(i,j)=psum(j)
              enddo 15
              y(i)=funk(psum)
            endif
          enddo 16
          iter=iter+ndim        Keep track of function evaluations.
          goto 1                Go back for the test of doneness and the next iteration.
        endif
      else
        iter=iter-1             Correct the evaluation count.
      endif
      goto 2
      END



      FUNCTION amotry(p,y,psum,mp,np,ndim,funk,ihi,fac)
      INTEGER ihi,mp,ndim,np,NMAX
      REAL amotry,fac,p(mp,np),psum(np),y(mp),funk
      PARAMETER (NMAX=20)
      EXTERNAL funk
C     USES funk
        Extrapolates by a factor fac through the face of the simplex across from the high point,
        tries it, and replaces the high point if the new point is better.
      INTEGER j
      REAL fac1,fac2,ytry,ptry(NMAX)
      fac1=(1.-fac)/ndim
      fac2=fac1-fac
      do 11 j=1,ndim
        ptry(j)=psum(j)*fac1-p(ihi,j)*fac2
      enddo 11
```

```
ytry=funk(ptry)              Evaluate the function at the trial point.
if (ytry.lt.y(ihi)) then     If it's better than the highest, then replace the highest.
    y(ihi)=ytry
    do 12 j=1,ndim
        psum(j)=psum(j)-p(ihi,j)+ptry(j)
        p(ihi,j)=ptry(j)
    enddo 12
endif
amotry=ytry
return
END
```

CITED REFERENCES AND FURTHER READING:

Nelder, J.A., and Mead, R. 1965, *Computer Journal*, vol. 7, pp. 308–313. [1]

Yarbro, L.A., and Deming, S.N. 1974, *Analytica Chimica Acta*, vol. 73, pp. 391–398.

Jacoby, S.L.S, Kowalik, J.S., and Pizzo, J.T. 1972, *Iterative Methods for Nonlinear Optimization Problems* (Englewood Cliffs, NJ: Prentice-Hall).

10.5 Direction Set (Powell's) Methods in Multidimensions

We know (§10.1–§10.3) how to minimize a function of one variable. If we start at a point $\mathbf{P}$ in N-dimensional space, and proceed from there in some vector direction $\mathbf{n}$, then any function of N variables $f(\mathbf{P})$ can be minimized along the line $\mathbf{n}$ by our one-dimensional methods. One can dream up various multidimensional minimization methods that consist of sequences of such line minimizations. Different methods will differ only by how, at each stage, they choose the next direction $\mathbf{n}$ to try. All such methods presume the existence of a "black-box" sub-algorithm, which we might call `linmin` (given as an explicit routine at the end of this section), whose definition can be taken for now as

> `linmin`: Given as input the vectors $\mathbf{P}$ and $\mathbf{n}$, and the function f, find the scalar λ that minimizes $f(\mathbf{P}+\lambda\mathbf{n})$. Replace $\mathbf{P}$ by $\mathbf{P}+\lambda\mathbf{n}$. Replace $\mathbf{n}$ by $\lambda\mathbf{n}$. Done.

All the minimization methods in this section and in the two sections following fall under this general schema of successive line minimizations. (The algorithm in §10.7 does not need very accurate line minimizations. Accordingly, it has its own approximate line minimization routine, `lnsrch`.) In this section we consider a class of methods whose choice of successive directions does not involve explicit computation of the function's gradient; the next two sections do require such gradient calculations. You will note that we need not specify whether `linmin` uses gradient information or not. That choice is up to you, and its optimization depends on your particular function. You would be crazy, however, to use gradients in `linmin` and *not* use them in the choice of directions, since in this latter role they can drastically reduce the total computational burden.

```
ytry=funk(ptry)              Evaluate the function at the trial point.
if (ytry.lt.y(ihi)) then     If it's better than the highest, then replace the highest.
    y(ihi)=ytry
    do 12 j=1,ndim
        psum(j)=psum(j)-p(ihi,j)+ptry(j)
        p(ihi,j)=ptry(j)
    enddo 12
endif
amotry=ytry
return
END
```

CITED REFERENCES AND FURTHER READING:

Nelder, J.A., and Mead, R. 1965, *Computer Journal*, vol. 7, pp. 308–313. [1]

Yarbro, L.A., and Deming, S.N. 1974, *Analytica Chimica Acta*, vol. 73, pp. 391–398.

Jacoby, S.L.S, Kowalik, J.S., and Pizzo, J.T. 1972, *Iterative Methods for Nonlinear Optimization Problems* (Englewood Cliffs, NJ: Prentice-Hall).

10.5 Direction Set (Powell's) Methods in Multidimensions

We know (§10.1–§10.3) how to minimize a function of one variable. If we start at a point $\mathbf{P}$ in N-dimensional space, and proceed from there in some vector direction $\mathbf{n}$, then any function of N variables $f(\mathbf{P})$ can be minimized along the line $\mathbf{n}$ by our one-dimensional methods. One can dream up various multidimensional minimization methods that consist of sequences of such line minimizations. Different methods will differ only by how, at each stage, they choose the next direction $\mathbf{n}$ to try. All such methods presume the existence of a "black-box" sub-algorithm, which we might call `linmin` (given as an explicit routine at the end of this section), whose definition can be taken for now as

> `linmin`: Given as input the vectors $\mathbf{P}$ and $\mathbf{n}$, and the function f, find the scalar λ that minimizes $f(\mathbf{P}+\lambda\mathbf{n})$. Replace $\mathbf{P}$ by $\mathbf{P}+\lambda\mathbf{n}$. Replace $\mathbf{n}$ by $\lambda\mathbf{n}$. Done.

All the minimization methods in this section and in the two sections following fall under this general schema of successive line minimizations. (The algorithm in §10.7 does not need very accurate line minimizations. Accordingly, it has its own approximate line minimization routine, `lnsrch`.) In this section we consider a class of methods whose choice of successive directions does not involve explicit computation of the function's gradient; the next two sections do require such gradient calculations. You will note that we need not specify whether `linmin` uses gradient information or not. That choice is up to you, and its optimization depends on your particular function. You would be crazy, however, to use gradients in `linmin` and *not* use them in the choice of directions, since in this latter role they can drastically reduce the total computational burden.

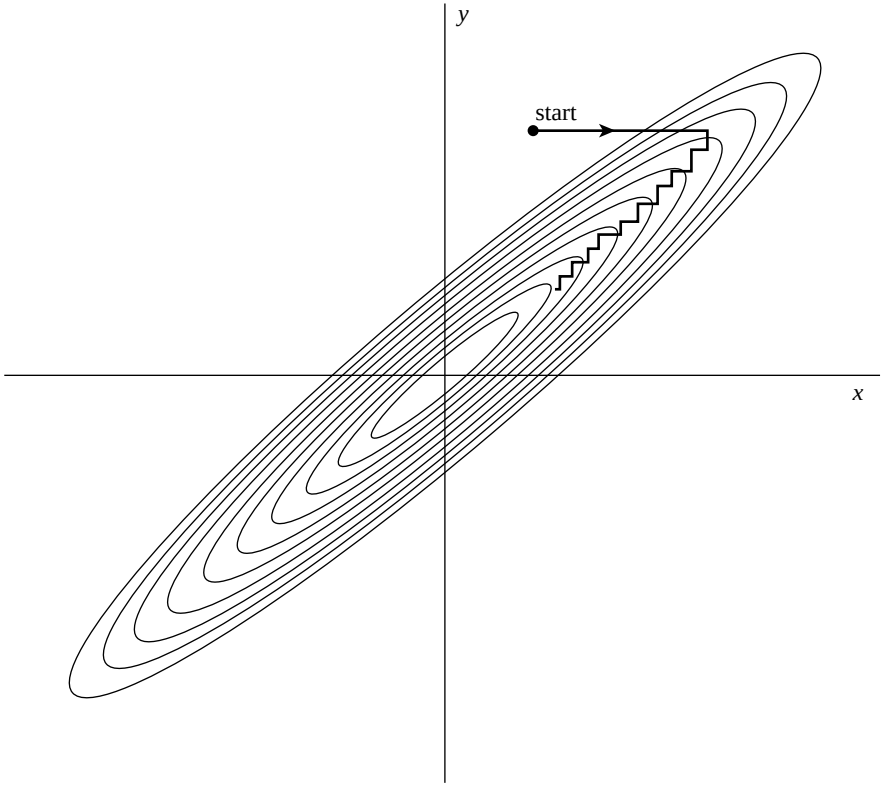


Figure 10.5.1. Successive minimizations along coordinate directions in a long, narrow "valley" (shown as contour lines). Unless the valley is optimally oriented, this method is extremely inefficient, taking many tiny steps to get to the minimum, crossing and re-crossing the principal axis.

But what if, in your application, calculation of the gradient is out of the question. You might first think of this simple method: Take the unit vectors $\mathbf{e}_1, \mathbf{e}_2, \ldots \mathbf{e}_N$ as a *set of directions*. Using `linmin`, move along the first direction to its minimum, then *from there* along the second direction to *its* minimum, and so on, cycling through the whole set of directions as many times as necessary, until the function stops decreasing.

This simple method is actually not too bad for many functions. Even more interesting is why it *is* bad, i.e. very inefficient, for some other functions. Consider a function of two dimensions whose contour map (level lines) happens to define a long, narrow valley at some angle to the coordinate basis vectors (see Figure 10.5.1). Then the only way "down the length of the valley" going along the basis vectors at each stage is by a series of many tiny steps. More generally, in N dimensions, if the function's second derivatives are much larger in magnitude in some directions than in others, then many cycles through all N basis vectors will be required in order to get anywhere. This condition is not all that unusual; according to Murphy's Law, you should count on it.

Obviously what we need is a better set of directions than the $\mathbf{e}_i$'s. All *direction set methods* consist of prescriptions for updating the set of directions as the method proceeds, attempting to come up with a set which either (i) includes some very

good directions that will take us far along narrow valleys, or else (more subtly) (ii) includes some number of "non-interfering" directions with the special property that minimization along one is not "spoiled" by subsequent minimization along another, so that interminable cycling through the set of directions can be avoided.

Conjugate Directions

This concept of "non-interfering" directions, more conventionally called *conjugate directions*, is worth making mathematically explicit.

First, note that if we minimize a function along some direction $\mathbf{u}$, then the gradient of the function must be perpendicular to $\mathbf{u}$ at the line minimum; if not, then there would still be a nonzero directional derivative along $\mathbf{u}$.

Next take some particular point $\mathbf{P}$ as the origin of the coordinate system with coordinates $\mathbf{x}$. Then any function f can be approximated by its Taylor series

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \cdots \\ &\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2}\, \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \end{aligned} \tag{10.5.1}$$

where

$$c \equiv f(\mathbf{P}) \qquad \mathbf{b} \equiv -\nabla f|_{\mathbf{P}} \qquad [\mathbf{A}]_{ij} \equiv \left. \frac{\partial^2 f}{\partial x_i \partial x_j} \right|_{\mathbf{P}} \tag{10.5.2}$$

The matrix $\mathbf{A}$ whose components are the second partial derivative matrix of the function is called the *Hessian matrix* of the function at $\mathbf{P}$.

In the approximation of (10.5.1), the gradient of f is easily calculated as

$$\nabla f = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{10.5.3}$$

(This implies that the gradient will vanish — the function will be at an extremum — at a value of $\mathbf{x}$ obtained by solving $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. This idea we will return to in §10.7!)

How does the gradient ∇f *change* as we move along some direction? Evidently

$$\delta(\nabla f) = \mathbf{A} \cdot (\delta \mathbf{x}) \tag{10.5.4}$$

Suppose that we have moved along some direction $\mathbf{u}$ to a minimum and now propose to move along some new direction $\mathbf{v}$. The condition that motion along $\mathbf{v}$ not *spoil* our minimization along $\mathbf{u}$ is just that the gradient stay perpendicular to $\mathbf{u}$, i.e., that the change in the gradient be perpendicular to $\mathbf{u}$. By equation (10.5.4) this is just

$$0 = \mathbf{u} \cdot \delta(\nabla f) = \mathbf{u} \cdot \mathbf{A} \cdot \mathbf{v} \tag{10.5.5}$$

When (10.5.5) holds for two vectors $\mathbf{u}$ and $\mathbf{v}$, they are said to be *conjugate*. When the relation holds pairwise for all members of a set of vectors, they are said to be a conjugate set. If you do successive line minimization of a function along a conjugate set of directions, then you don't need to redo any of those directions

(unless, of course, you spoil things by minimizing along a direction that they are *not* conjugate to).

A triumph for a direction set method is to come up with a set of N linearly independent, mutually conjugate directions. Then, one pass of N line minimizations will put it exactly at the minimum of a quadratic form like (10.5.1). For functions f that are not exactly quadratic forms, it won't be exactly at the minimum; but repeated cycles of N line minimizations will in due course converge *quadratically* to the minimum.

Powell's Quadratically Convergent Method

Powell first discovered a direction set method that does produce N mutually conjugate directions. Here is how it goes: Initialize the set of directions $\mathbf{u}_i$ to the basis vectors,

$$\mathbf{u}_i = \mathbf{e}_i \qquad i = 1, \ldots, N \tag{10.5.6}$$

Now repeat the following sequence of steps ("basic procedure") until your function stops decreasing:

- Save your starting position as $\mathbf{P}_0$.
- For $i = 1, \ldots, N$, move $\mathbf{P}_{i-1}$ to the minimum along direction $\mathbf{u}_i$ and call this point $\mathbf{P}_i$.
- For $i = 1, \ldots, N-1$, set $\mathbf{u}_i \leftarrow \mathbf{u}_{i+1}$.
- Set $\mathbf{u}_N \leftarrow \mathbf{P}_N - \mathbf{P}_0$.
- Move $\mathbf{P}_N$ to the minimum along direction $\mathbf{u}_N$ and call this point $\mathbf{P}_0$.

Powell, in 1964, showed that, for a quadratic form like (10.5.1), k iterations of the above basic procedure produce a set of directions $\mathbf{u}_i$ whose last k members are mutually conjugate. Therefore, N iterations of the basic procedure, amounting to $N(N+1)$ line minimizations in all, will exactly minimize a quadratic form. Brent [1] gives proofs of these statements in accessible form.

Unfortunately, there is a problem with Powell's quadratically convergent algorithm. The procedure of throwing away, at each stage, $\mathbf{u}_1$ in favor of $\mathbf{P}_N - \mathbf{P}_0$ tends to produce sets of directions that "fold up on each other" and become linearly dependent. Once this happens, then the procedure finds the minimum of the function f only over a subspace of the full N-dimensional case; in other words, it gives the wrong answer. Therefore, the algorithm must not be used in the form given above.

There are a number of ways to fix up the problem of linear dependence in Powell's algorithm, among them:

1. You can reinitialize the set of directions $\mathbf{u}_i$ to the basis vectors $\mathbf{e}_i$ after every N or $N+1$ iterations of the basic procedure. This produces a serviceable method, which we commend to you if quadratic convergence is important for your application (i.e., if your functions are close to quadratic forms and if you desire high accuracy).

2. Brent points out that the set of directions can equally well be reset to the columns of any orthogonal matrix. Rather than throw away the information on conjugate directions already built up, he resets the direction set to calculated principal directions of the matrix $\mathbf{A}$ (which he gives a procedure for determining).

The calculation is essentially a singular value decomposition algorithm (see §2.6). Brent has a number of other cute tricks up his sleeve, and his modification of Powell's method is probably the best presently known. Consult [1] for a detailed description and listing of the program. Unfortunately it is rather too elaborate for us to include here.

3. You can give up the property of quadratic convergence in favor of a more heuristic scheme (due to Powell) which tries to find a few good directions along narrow valleys instead of N necessarily conjugate directions. This is the method that we now implement. (It is also the version of Powell's method given in Acton [2], from which parts of the following discussion are drawn.)

Discarding the Direction of Largest Decrease

The fox and the grapes: Now that we are going to give up the property of quadratic convergence, was it so important after all? That depends on the function that you are minimizing. Some applications produce functions with long, twisty valleys. Quadratic convergence is of no particular advantage to a program which must slalom down the length of a valley floor that twists one way and another (and another, and another, ... – there are N dimensions!). Along the long direction, a quadratically convergent method is trying to extrapolate to the minimum of a parabola which just isn't (yet) there; while the conjugacy of the $N-1$ transverse directions keeps getting spoiled by the twists.

Sooner or later, however, we do arrive at an approximately ellipsoidal minimum (cf. equation 10.5.1 when **b**, the gradient, is zero). Then, depending on how much accuracy we require, a method with quadratic convergence can save us several times N^2 extra line minimizations, since quadratic convergence *doubles* the number of significant figures at each iteration.

The basic idea of our now-modified Powell's method is still to take $\mathbf{P}_N - \mathbf{P}_0$ as a new direction; it is, after all, the average direction moved after trying all N possible directions. For a valley whose long direction is twisting slowly, this direction is likely to give us a good run along the new long direction. The change is to discard the old direction along which the function f made its *largest decrease*. This seems paradoxical, since that direction was the *best* of the previous iteration. However, it is also likely to be a major component of the new direction that we are adding, so dropping it gives us the best chance of avoiding a buildup of linear dependence.

There are a couple of exceptions to this basic idea. Sometimes it is better *not* to add a new direction at all. Define

$$f_0 \equiv f(\mathbf{P}_0) \qquad f_N \equiv f(\mathbf{P}_N) \qquad f_E \equiv f(2\mathbf{P}_N - \mathbf{P}_0) \tag{10.5.7}$$

Here f_E is the function value at an "extrapolated" point somewhat further along the proposed new direction. Also define Δf to be the magnitude of the largest decrease along one particular direction of the present basic procedure iteration. (Δf is a positive number.) Then:

1. If $f_E \geq f_0$, then keep the old set of directions for the next basic procedure, because the average direction $\mathbf{P}_N - \mathbf{P}_0$ is all played out.

2. If $2\,(f_0 - 2f_N + f_E)\,[(f_0 - f_N) - \Delta f]^2 \geq (f_0 - f_E)^2 \Delta f$, then keep the old set of directions for the next basic procedure, because either (i) the decrease along

the average direction was not primarily due to any single direction's decrease, or (ii) there is a substantial second derivative along the average direction and we seem to be near to the bottom of its minimum.

The following routine implements Powell's method in the version just described. In the routine, `xi` is the matrix whose columns are the set of directions $\mathbf{n}_i$; otherwise the correspondence of notation should be self-evident.

```
      SUBROUTINE powell(p,xi,n,np,ftol,iter,fret)
      INTEGER iter,n,np,NMAX,ITMAX
      REAL fret,ftol,p(np),xi(np,np),func,TINY
      EXTERNAL func
      PARAMETER (NMAX=20,ITMAX=200,TINY=1.e-25)
C     USES func,linmin
        Minimization of a function func of n variables. (func is not an argument, it is a fixed function name.) Input consists of an initial starting point p(1:n); an initial matrix xi(1:n,1:n) with physical dimensions np by np, and whose columns contain the initial set of directions (usually the n unit vectors); and ftol, the fractional tolerance in the function value such that failure to decrease by more than this amount on one iteration signals doneness. On output, p is set to the best point found, xi is the then-current direction set, fret is the returned function value at p, and iter is the number of iterations taken. The routine linmin is used.
        Parameters: Maximum value of n, maximum allowed iterations, and a small number.
      INTEGER i,ibig,j
      REAL del,fp,fptt,t,pt(NMAX),ptt(NMAX),xit(NMAX)
      fret=func(p)
      do 11 j=1,n                             Save the initial point.
        pt(j)=p(j)
      enddo 11
      iter=0
1     iter=iter+1
      fp=fret
      ibig=0
      del=0.                                  Will be the biggest function decrease.
      do 13 i=1,n                             In each iteration, loop over all directions in the set.
        do 12 j=1,n                           Copy the direction,
          xit(j)=xi(j,i)
        enddo 12
        fptt=fret
        call linmin(p,xit,n,fret)             minimize along it,
        if(fptt-fret.gt.del)then              and record it if it is the largest decrease so far.
          del=fptt-fret
          ibig=i
        endif
      enddo 13
      if(2.*(fp-fret).le.ftol*(abs(fp)+abs(fret))+TINY)return    Termination criterion.
      if(iter.eq.ITMAX) pause 'powell exceeding maximum iterations'
      do 14 j=1,n                             Construct the extrapolated point and the average direction moved. Save the old starting point.
        ptt(j)=2.*p(j)-pt(j)
        xit(j)=p(j)-pt(j)
        pt(j)=p(j)
      enddo 14
      fptt=func(ptt)                          Function value at extrapolated point.
      if(fptt.ge.fp)goto 1                    One reason not to use new direction.
      t=2.*(fp-2.*fret+fptt)*(fp-fret-del)**2-del*(fp-fptt)**2
      if(t.ge.0.)goto 1                       Other reason not to use new direction.
      call linmin(p,xit,n,fret)               Move to the minimum of the new direction,
      do 15 j=1,n                             and save the new direction.
        xi(j,ibig)=xi(j,n)
        xi(j,n)=xit(j)
      enddo 15
      goto 1                                  Back for another iteration.
      END
```

Implementation of Line Minimization

In the above routine, you might have wondered why we didn't make the function name `func` an argument of the routine. The reason is buried in a slightly dirty FORTRAN practicality in our implementation of `linmin`.

Make no mistake, there is a *right* way to implement `linmin`: It is to use the *methods* of one-dimensional minimization described in §10.1–§10.3, but to rewrite the programs of those sections so that their bookkeeping is done on vector-valued points **P** (all lying along a given direction **n**) rather than scalar-valued abscissas x. That straightforward task produces long routines densely populated with "`do k=1,n`" loops.

We do not have space to include such routines in this book. Our `linmin`, which works just fine, is instead a kind of bookkeeping swindle. It constructs an "artificial" function of one variable called `f1dim`, which is the value of your function `func` along the line going through the point `p` in the direction `xi`. `linmin` communicates with `f1dim` through a common block. It then calls our familiar one-dimensional routines `mnbrak` (§10.1) and `brent` (§10.2) and instructs them to minimize `f1dim`.

Still following? Then try this: `brent` receives the function name `f1dim`, which it dutifully calls. But there is no way to signal to `f1dim` that it is supposed to use your function name, which could have been passed to `linmin` as an argument. Therefore, we have to make `f1dim` use a *fixed* function name, namely `func`. The situation is reminiscent of Henry Ford's black automobile: `powell` will minimize any function, as long as it is named `func`. Needed to remedy this situation is a way to pass a function name through a common block; this is lacking in FORTRAN.

The only thing inefficient about `linmin` is this: Its use as an interface between a multidimensional minimization strategy and a one-dimensional minimization routine results in some unnecessary copying of vectors hither and yon. That should not normally be a significant addition to the overall computational burden, but we cannot disguise its inelegance.

```
      SUBROUTINE linmin(p,xi,n,fret)
      INTEGER n,NMAX
      REAL fret,p(n),xi(n),TOL
      PARAMETER (NMAX=50,TOL=1.e-4)          Maximum anticipated n, and TOL passed to brent.
C     USES brent,f1dim,mnbrak
        Given an n-dimensional point p(1:n) and an n-dimensional direction xi(1:n), moves and
        resets p to where the function func(p) takes on a minimum along the direction xi from
        p, and replaces xi by the actual vector displacement that p was moved. Also returns as
        fret the value of func at the returned location p. This is actually all accomplished by
        calling the routines mnbrak and brent.
      INTEGER j,ncom
      REAL ax,bx,fa,fb,fx,xmin,xx,pcom(NMAX),xicom(NMAX),brent
      COMMON /f1com/ pcom,xicom,ncom
      EXTERNAL f1dim
      ncom=n                                 Set up the common block.
      do 11 j=1,n
        pcom(j)=p(j)
        xicom(j)=xi(j)
      enddo 11
      ax=0.                                  Initial guess for brackets.
      xx=1.
      call mnbrak(ax,xx,bx,fa,fx,fb,f1dim)
      fret=brent(ax,xx,bx,f1dim,TOL,xmin)
      do 12 j=1,n                            Construct the vector results to return.
```

```
        xi(j)=xmin*xi(j)
        p(j)=p(j)+xi(j)
    enddo 12
    return
    END


    FUNCTION f1dim(x)
    INTEGER NMAX
    REAL f1dim,func,x
    PARAMETER (NMAX=50)
C   USES func
        Used by linmin as the function passed to mnbrak and brent.
    INTEGER j,ncom
    REAL pcom(NMAX),xicom(NMAX),xt(NMAX)
    COMMON /f1com/ pcom,xicom,ncom
    do 11 j=1,ncom
        xt(j)=pcom(j)+x*xicom(j)
    enddo 11
    f1dim=func(xt)
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 7. [1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 464–467. [2]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), pp. 259–262.

10.6 Conjugate Gradient Methods in Multidimensions

We consider now the case where you are able to calculate, at a given N-dimensional point $\mathbf{P}$, not just the value of a function $f(\mathbf{P})$ but also the gradient (vector of first partial derivatives) $\nabla f(\mathbf{P})$.

A rough counting argument will show how advantageous it is to use the gradient information: Suppose that the function f is roughly approximated as a quadratic form, as above in equation (10.5.1),

$$f(\mathbf{x}) \approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2}\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \tag{10.6.1}$$

Then the number of unknown parameters in f is equal to the number of free parameters in $\mathbf{A}$ and $\mathbf{b}$, which is $\frac{1}{2}N(N+1)$, which we see to be of order N^2. Changing any one of these parameters can move the location of the minimum. Therefore, we should not expect to be able to *find* the minimum until we have collected an equivalent information content, of order N^2 numbers.

```
        xi(j)=xmin*xi(j)
        p(j)=p(j)+xi(j)
    enddo 12
    return
    END


    FUNCTION f1dim(x)
    INTEGER NMAX
    REAL f1dim,func,x
    PARAMETER (NMAX=50)
C   USES func
        Used by linmin as the function passed to mnbrak and brent.
    INTEGER j,ncom
    REAL pcom(NMAX),xicom(NMAX),xt(NMAX)
    COMMON /f1com/ pcom,xicom,ncom
    do 11 j=1,ncom
        xt(j)=pcom(j)+x*xicom(j)
    enddo 11
    f1dim=func(xt)
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 7. [1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 464–467. [2]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), pp. 259–262.

10.6 Conjugate Gradient Methods in Multidimensions

We consider now the case where you are able to calculate, at a given N-dimensional point $\mathbf{P}$, not just the value of a function $f(\mathbf{P})$ but also the gradient (vector of first partial derivatives) $\nabla f(\mathbf{P})$.

A rough counting argument will show how advantageous it is to use the gradient information: Suppose that the function f is roughly approximated as a quadratic form, as above in equation (10.5.1),

$$f(\mathbf{x}) \approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2}\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \tag{10.6.1}$$

Then the number of unknown parameters in f is equal to the number of free parameters in $\mathbf{A}$ and $\mathbf{b}$, which is $\frac{1}{2}N(N+1)$, which we see to be of order N^2. Changing any one of these parameters can move the location of the minimum. Therefore, we should not expect to be able to *find* the minimum until we have collected an equivalent information content, of order N^2 numbers.

In the direction set methods of §10.5, we collected the necessary information by making on the order of N^2 separate line minimizations, each requiring "a few" (but sometimes a *big* few!) function evaluations. Now, each evaluation of the gradient will bring us N new components of information. If we use them wisely, we should need to make only of order N separate line minimizations. That is in fact the case for the algorithms in this section and the next.

A factor of N improvement in computational speed is not necessarily implied. As a rough estimate, we might imagine that the calculation of *each component* of the gradient takes about as long as evaluating the function itself. In that case there will be of order N^2 equivalent function evaluations both with and without gradient information. Even if the advantage is not of order N, however, it is nevertheless quite substantial: (i) Each calculated component of the gradient will typically save not just one function evaluation, but a number of them, equivalent to, say, a whole line minimization. (ii) There is often a high degree of redundancy in the formulas for the various components of a function's gradient; when this is so, especially when there is also redundancy with the calculation of the function, then the calculation of the gradient may cost significantly less than N function evaluations.

A common beginner's error is to assume that any reasonable way of incorporating gradient information should be about as good as any other. This line of thought leads to the following *not very good* algorithm, the *steepest descent method*:

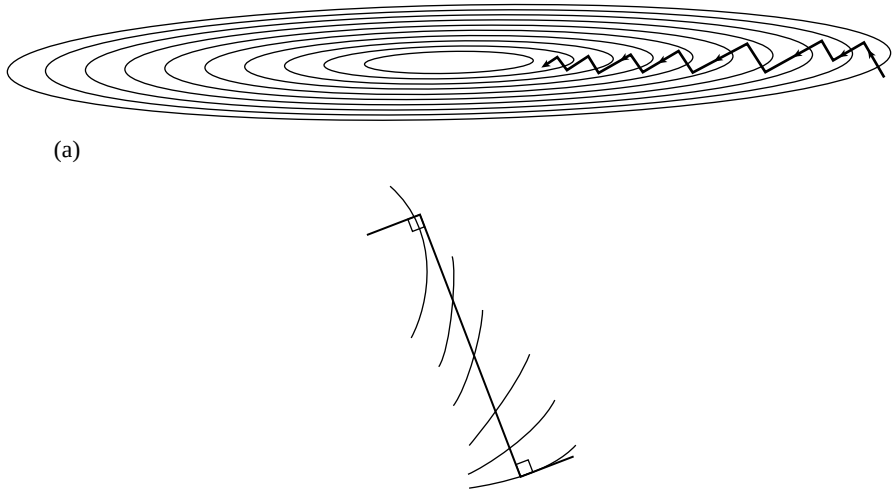
> Steepest Descent: Start at a point $\mathbf{P}_0$. As many times as needed, move from point $\mathbf{P}_i$ to the point $\mathbf{P}_{i+1}$ by minimizing along the line from $\mathbf{P}_i$ in the direction of the local downhill gradient $-\nabla f(\mathbf{P}_i)$.

The problem with the steepest descent method (which, incidentally, goes back to Cauchy), is similar to the problem that was shown in Figure 10.5.1. The method will perform many small steps in going down a long, narrow valley, even if the valley is a perfect quadratic form. You might have hoped that, say in two dimensions, your first step would take you to the valley floor, the second step directly down the long axis; but remember that the new gradient at the minimum point of any line minimization is perpendicular to the direction just traversed. Therefore, with the steepest descent method, you *must* make a right angle turn, which does *not*, in general, take you to the minimum. (See Figure 10.6.1.)

Just as in the discussion that led up to equation (10.5.5), we really want a way of proceeding not down the new gradient, but rather in a direction that is somehow constructed to be *conjugate* to the old gradient, and, insofar as possible, to all previous directions traversed. Methods that accomplish this construction are called *conjugate gradient* methods.

In §2.7 we discussed the conjugate gradient method as a technique for solving linear algebraic equations by minimizing a quadratic form. That formalism can also be applied to the problem of minimizing a function *approximated* by the quadratic form (10.6.1). Recall that, starting with an arbitrary initial vector $\mathbf{g}_0$ and letting $\mathbf{h}_0 = \mathbf{g}_0$, the conjugate gradient method constructs two sequences of vectors from the recurrence

$$\mathbf{g}_{i+1} = \mathbf{g}_i - \lambda_i \mathbf{A} \cdot \mathbf{h}_i \qquad \mathbf{h}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i \qquad i = 0, 1, 2, \ldots \tag{10.6.2}$$



(a)

(b)

Figure 10.6.1. (a) Steepest descent method in a long, narrow "valley." While more efficient than the strategy of Figure 10.5.1, steepest descent is nonetheless an inefficient strategy, taking many steps to reach the valley floor. (b) Magnified view of one step: A step starts off in the local gradient direction, perpendicular to the contour lines, and traverses a straight line until a local minimum is reached, where the traverse is parallel to the local contour lines.

The vectors satisfy the orthogonality and conjugacy conditions

$$\mathbf{g}_i \cdot \mathbf{g}_j = 0 \qquad \mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_j = 0 \qquad \mathbf{g}_i \cdot \mathbf{h}_j = 0 \qquad j < i \tag{10.6.3}$$

The scalars λ_i and γ_i are given by

$$\lambda_i = \frac{\mathbf{g}_i \cdot \mathbf{g}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} = \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \tag{10.6.4}$$

$$\gamma_i = \frac{\mathbf{g}_{i+1} \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.6.5}$$

Equations (10.6.2)–(10.6.5) are simply equations (2.7.32)–(2.7.35) for a symmetric $\mathbf{A}$ in a new notation. (A self-contained derivation of these results in the context of function minimization is given by Polak [1].)

Now suppose that we knew the Hessian matrix $\mathbf{A}$ in equation (10.6.1). Then we could use the construction (10.6.2) to find successively conjugate directions $\mathbf{h}_i$ along which to line-minimize. After N such, we would efficiently have arrived at the minimum of the quadratic form. But we don't know $\mathbf{A}$.

Here is a remarkable theorem to save the day: Suppose we happen to have $\mathbf{g}_i = -\nabla f(\mathbf{P}_i)$, for some point $\mathbf{P}_i$, where f is of the form (10.6.1). Suppose that we proceed from $\mathbf{P}_i$ along the direction $\mathbf{h}_i$ to the local minimum of f located at some point $\mathbf{P}_{i+1}$ and then set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{P}_{i+1})$. Then, this $\mathbf{g}_{i+1}$ is the same vector as would have been constructed by equation (10.6.2). (And we have constructed it without knowledge of $\mathbf{A}$!)

Proof: By equation (10.5.3), $\mathbf{g}_i = -\mathbf{A} \cdot \mathbf{P}_i + \mathbf{b}$, and

$$\mathbf{g}_{i+1} = -\mathbf{A} \cdot (\mathbf{P}_i + \lambda \mathbf{h}_i) + \mathbf{b} = \mathbf{g}_i - \lambda \mathbf{A} \cdot \mathbf{h}_i \tag{10.6.6}$$

with λ chosen to take us to the line minimum. But at the line minimum $\mathbf{h}_i \cdot \nabla f = -\mathbf{h}_i \cdot \mathbf{g}_{i+1} = 0$. This latter condition is easily combined with (10.6.6) to solve for λ. The result is exactly the expression (10.6.4). But with this value of λ, (10.6.6) is the same as (10.6.2), q.e.d.

We have, then, the basis of an algorithm that requires neither knowledge of the Hessian matrix **A**, nor even the storage necessary to store such a matrix. A sequence of directions $\mathbf{h}_i$ is constructed, using only line minimizations, evaluations of the gradient vector, and an auxiliary vector to store the latest in the sequence of **g**'s.

The algorithm described so far is the original Fletcher-Reeves version of the conjugate gradient algorithm. Later, Polak and Ribiere introduced one tiny, but sometimes significant, change. They proposed using the form

$$\gamma_i = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i) \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.6.7}$$

instead of equation (10.6.5). "Wait," you say, "aren't they equal by the orthogonality conditions (10.6.3)?" They are equal for exact quadratic forms. In the real world, however, your function is not exactly a quadratic form. Arriving at the supposed minimum of the quadratic form, you may still need to proceed for another set of iterations. There is some evidence [2] that the Polak-Ribiere formula accomplishes the transition to further iterations more gracefully: When it runs out of steam, it tends to reset **h** to be down the local gradient, which is equivalent to beginning the conjugate-gradient procedure anew.

The following routine implements the Polak-Ribiere variant, which we recommend; but changing one program line, as shown, will give you Fletcher-Reeves. The routine presumes the existence of a function `func(p)`, where `p(1:n)` is a vector of length `n`, and also presumes the existence of a subroutine `dfunc(p,df)` that returns the vector gradient `df(1:n)` evaluated at the input point `p`.

The routine calls `linmin` to do the line minimizations. As already discussed, you may wish to use a modified version of `linmin` that uses `dbrent` instead of `brent`, i.e., that uses the gradient in doing the line minimizations. See note below.

```
      SUBROUTINE frprmn(p,n,ftol,iter,fret)
      INTEGER iter,n,NMAX,ITMAX
      REAL fret,ftol,p(n),EPS,func
      EXTERNAL func
      PARAMETER (NMAX=50,ITMAX=200,EPS=1.e-10)
C     USES dfunc,func,linmin
```

Given a starting point `p` that is a vector of length `n`, Fletcher-Reeves-Polak-Ribiere minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence tolerance on the function value is input as `ftol`. Returned quantities are `p` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `linmin` is called to perform line minimizations.

Parameters: `NMAX` is the maximum anticipated value of `n`; `ITMAX` is the maximum allowed number of iterations; `EPS` is a small number to rectify special case of converging to exactly zero function value.

```
      INTEGER its,j
      REAL dgg,fp,gam,gg,g(NMAX),h(NMAX),xi(NMAX)
      fp=func(p)                              Initializations.
      call dfunc(p,xi)
      do 11 j=1,n
        g(j)=-xi(j)
        h(j)=g(j)
```

```
        xi(j)=h(j)
      enddo 11
      do 14 its=1,ITMAX                    Loop over iterations.
        iter=its
        call linmin(p,xi,n,fret)           Next statement is the normal return:
        if(2.*abs(fret-fp).le.ftol*(abs(fret)+abs(fp)+EPS))return
        fp=fret
        call dfunc(p,xi)
        gg=0.
        dgg=0.
        do 12 j=1,n
          gg=gg+g(j)**2
C         dgg=dgg+xi(j)**2                 This statement for Fletcher-Reeves.
          dgg=dgg+(xi(j)+g(j))*xi(j)       This statement for Polak-Ribiere.
        enddo 12
        if(gg.eq.0.)return                 Unlikely. If gradient is exactly zero then we are al-
        gam=dgg/gg                             ready done.
        do 13 j=1,n
          g(j)=-xi(j)
          h(j)=g(j)+gam*h(j)
          xi(j)=h(j)
        enddo 13
      enddo 14
      pause 'frprmn maximum iterations exceeded'
      return
      END
```

Note on Line Minimization Using Derivatives

Kindly reread the last part of §10.5. We here want to do the same thing, but using derivative information in performing the line minimization.

Rather than reprint the whole routine `linmin` just to show one modified statement, let us just tell you what the change is: The statement

```
      fret=brent(ax,xx,bx,f1dim,tol,xmin)
```

should be replaced by

```
      fret=dbrent(ax,xx,bx,f1dim,df1dim,tol,xmin)
```

You must also include the following function, which is analogous to `f1dim` as discussed in §10.5. And remember, your function must be named `func`, and its gradient calculation must be named `dfunc`.

```
      FUNCTION df1dim(x)
      INTEGER NMAX
      REAL df1dim,x
      PARAMETER (NMAX=50)
C     USES dfunc
      INTEGER j,ncom
      REAL df(NMAX),pcom(NMAX),xicom(NMAX),xt(NMAX)
      COMMON /f1com/ pcom,xicom,ncom
      do 11 j=1,ncom
        xt(j)=pcom(j)+x*xicom(j)
      enddo 11
      call dfunc(xt,df)
      df1dim=0.
```

```
do 12 j=1,ncom
    df1dim=df1dim+df(j)*xicom(j)
enddo 12
return
END
```

CITED REFERENCES AND FURTHER READING:

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), §2.3. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.7 (by K.W. Brodlie). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §8.7.

10.7 Variable Metric Methods in Multidimensions

The goal of *variable metric* methods, which are sometimes called *quasi-Newton* methods, is not different from the goal of conjugate gradient methods: to accumulate information from successive line minimizations so that N such line minimizations lead to the exact minimum of a quadratic form in N dimensions. In that case, the method will also be quadratically convergent for more general smooth functions.

Both variable metric and conjugate gradient methods require that you are able to compute your function's gradient, or first partial derivatives, at arbitrary points. The variable metric approach differs from the conjugate gradient in the way that it stores and updates the information that is accumulated. Instead of requiring intermediate storage on the order of N, the number of dimensions, it requires a matrix of size $N \times N$. Generally, for any moderate N, this is an entirely trivial disadvantage.

On the other hand, there is not, as far as we know, any overwhelming advantage that the variable metric methods hold over the conjugate gradient techniques, except perhaps a historical one. Developed somewhat earlier, and more widely propagated, the variable metric methods have by now developed a wider constituency of satisfied users. Likewise, some fancier implementations of variable metric methods (going beyond the scope of this book, see below) have been developed to a greater level of sophistication on issues like the minimization of roundoff error, handling of special conditions, and so on. *We* tend to use variable metric rather than conjugate gradient, but we have no reason to urge this habit on you.

Variable metric methods come in two main flavors. One is the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to as simply *Fletcher-Powell*). The other goes by the name *Broyden-Fletcher-Goldfarb-Shanno (BFGS)*. The BFGS and DFP schemes differ only in details of their roundoff error, convergence tolerances, and similar "dirty" issues which are outside of our scope [1,2]. However, it has become generally recognized that, empirically, the BFGS scheme is superior in these details. We will implement BFGS in this section.

As before, we imagine that our arbitrary function $f(\mathbf{x})$ can be locally approximated by the quadratic form of equation (10.6.1). We don't, however, have any

```
do 12 j=1,ncom
    df1dim=df1dim+df(j)*xicom(j)
enddo 12
return
END
```

CITED REFERENCES AND FURTHER READING:

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), §2.3. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.7 (by K.W. Brodlie). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §8.7.

10.7 Variable Metric Methods in Multidimensions

The goal of *variable metric* methods, which are sometimes called *quasi-Newton* methods, is not different from the goal of conjugate gradient methods: to accumulate information from successive line minimizations so that N such line minimizations lead to the exact minimum of a quadratic form in N dimensions. In that case, the method will also be quadratically convergent for more general smooth functions.

Both variable metric and conjugate gradient methods require that you are able to compute your function's gradient, or first partial derivatives, at arbitrary points. The variable metric approach differs from the conjugate gradient in the way that it stores and updates the information that is accumulated. Instead of requiring intermediate storage on the order of N, the number of dimensions, it requires a matrix of size $N \times N$. Generally, for any moderate N, this is an entirely trivial disadvantage.

On the other hand, there is not, as far as we know, any overwhelming advantage that the variable metric methods hold over the conjugate gradient techniques, except perhaps a historical one. Developed somewhat earlier, and more widely propagated, the variable metric methods have by now developed a wider constituency of satisfied users. Likewise, some fancier implementations of variable metric methods (going beyond the scope of this book, see below) have been developed to a greater level of sophistication on issues like the minimization of roundoff error, handling of special conditions, and so on. *We* tend to use variable metric rather than conjugate gradient, but we have no reason to urge this habit on you.

Variable metric methods come in two main flavors. One is the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to as simply *Fletcher-Powell*). The other goes by the name *Broyden-Fletcher-Goldfarb-Shanno (BFGS)*. The BFGS and DFP schemes differ only in details of their roundoff error, convergence tolerances, and similar "dirty" issues which are outside of our scope [1,2]. However, it has become generally recognized that, empirically, the BFGS scheme is superior in these details. We will implement BFGS in this section.

As before, we imagine that our arbitrary function $f(\mathbf{x})$ can be locally approximated by the quadratic form of equation (10.6.1). We don't, however, have any

information about the values of the quadratic form's parameters $\mathbf{A}$ and $\mathbf{b}$, except insofar as we can glean such information from our function evaluations and line minimizations.

The basic idea of the variable metric method is to build up, iteratively, a good approximation to the inverse Hessian matrix $\mathbf{A}^{-1}$, that is, to construct a sequence of matrices $\mathbf{H}_i$ with the property,

$$\lim_{i\to\infty} \mathbf{H}_i = \mathbf{A}^{-1} \tag{10.7.1}$$

Even better if the limit is achieved after N iterations instead of ∞.

The reason that variable metric methods are sometimes called quasi-Newton methods can now be explained. Consider finding a minimum by using Newton's method to search for a zero of the gradient of the function. Near the current point $\mathbf{x}_i$, we have to second order

$$f(\mathbf{x}) = f(\mathbf{x}_i) + (\mathbf{x}-\mathbf{x}_i)\cdot\nabla f(\mathbf{x}_i) + \tfrac{1}{2}(\mathbf{x}-\mathbf{x}_i)\cdot\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i) \tag{10.7.2}$$

so

$$\nabla f(\mathbf{x}) = \nabla f(\mathbf{x}_i) + \mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i) \tag{10.7.3}$$

In Newton's method we set $\nabla f(\mathbf{x}) = 0$ to determine the next iteration point:

$$\mathbf{x} - \mathbf{x}_i = -\mathbf{A}^{-1}\cdot\nabla f(\mathbf{x}_i) \tag{10.7.4}$$

The left-hand side is the finite step we need take to get to the exact minimum; the right-hand side is known once we have accumulated an accurate $\mathbf{H} \approx \mathbf{A}^{-1}$.

The "quasi" in quasi-Newton is because we don't use the actual Hessian matrix of f, but instead use our current approximation of it. This is often *better* than using the true Hessian. We can understand this paradoxical result by considering the *descent directions* of f at $\mathbf{x}_i$. These are the directions $\mathbf{p}$ along which f decreases: $\nabla f\cdot\mathbf{p} < 0$. For the Newton direction (10.7.4) to be a descent direction, we must have

$$\nabla f(\mathbf{x}_i)\cdot(\mathbf{x}-\mathbf{x}_i) = -(\mathbf{x}-\mathbf{x}_i)\cdot\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i) < 0 \tag{10.7.5}$$

that is, $\mathbf{A}$ must be positive definite. In general, far from a minimum, we have no guarantee that the Hessian is positive definite. Taking the actual Newton step with the real Hessian can move us to points where the function is *increasing* in value. The idea behind quasi-Newton methods is to start with a positive definite, symmetric approximation to $\mathbf{A}$ (usually the unit matrix) and build up the approximating $\mathbf{H}_i$'s in such a way that the matrix $\mathbf{H}_i$ remains positive definite and symmetric. Far from the minimum, this guarantees that we always move in a downhill direction. Close to the minimum, the updating formula approaches the true Hessian and we enjoy the quadratic convergence of Newton's method.

When we are not close enough to the minimum, taking the full Newton step $\mathbf{p}$ even with a positive definite $\mathbf{A}$ need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. Once again we can use the backtracking strategy described in §9.7 to choose a step along the *direction* of the Newton step $\mathbf{p}$, but not necessarily all the way.

We won't rigorously derive the DFP algorithm for taking $\mathbf{H}_i$ into $\mathbf{H}_{i+1}$; you can consult [3] for clear derivations. Following Brodlie (in [2]), we will give the following heuristic motivation of the procedure.

Subtracting equation (10.7.4) at $\mathbf{x}_{i+1}$ from that same equation at $\mathbf{x}_i$ gives

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{A}^{-1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.6}$$

where $\nabla f_j \equiv \nabla f(\mathbf{x}_j)$. Having made the step from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$, we might reasonably want to require that the new approximation $\mathbf{H}_{i+1}$ satisfy (10.7.6) as if it were actually $\mathbf{A}^{-1}$, that is,

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}_{i+1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.7}$$

We might also imagine that the updating formula should be of the form $\mathbf{H}_{i+1} = \mathbf{H}_i + \text{correction}$.

What "objects" are around out of which to construct a correction term? Most notable are the two vectors $\mathbf{x}_{i+1} - \mathbf{x}_i$ and $\nabla f_{i+1} - \nabla f_i$; and there is also $\mathbf{H}_i$. There are not infinitely many natural ways of making a matrix out of these objects, especially if (10.7.7) must hold! One such way, the *DFP updating formula*, is

$$\begin{aligned} \mathbf{H}_{i+1} = \mathbf{H}_i &+ \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \otimes (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ &- \frac{[\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)] \otimes [\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \end{aligned} \tag{10.7.8}$$

where $\otimes$ denotes the "outer" or "direct" product of two vectors, a matrix: The ij component of $\mathbf{u} \otimes \mathbf{v}$ is $u_i v_j$. (You might want to verify that 10.7.8 does satisfy 10.7.7.)

The *BFGS updating formula* is exactly the same, but with one additional term,

$$\cdots + [(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]\, \mathbf{u} \otimes \mathbf{u} \tag{10.7.9}$$

where $\mathbf{u}$ is defined as the vector

$$\begin{aligned} \mathbf{u} \equiv &\frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ &- \frac{\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \end{aligned} \tag{10.7.10}$$

(You might also verify that this satisfies 10.7.7.)

You will have to take on faith — or else consult [3] for details of — the "deep" result that equation (10.7.8), with or without (10.7.9), does in fact converge to $\mathbf{A}^{-1}$ in N steps, if f is a quadratic form.

Here now is the routine `dfpmin` that implements the quasi-Newton method, and uses `lnsrch` from §9.7. As mentioned at the end of `newt` in §9.7, this algorithm can fail if your variables are badly scaled.

```
      SUBROUTINE dfpmin(p,n,gtol,iter,fret,func,dfunc)
      INTEGER iter,n,NMAX,ITMAX
      REAL fret,gtol,p(n),func,EPS,STPMX,TOLX
      PARAMETER (NMAX=50,ITMAX=200,STPMX=100.,EPS=3.e-8,TOLX=4.*EPS)
      EXTERNAL dfunc,func
C     USES dfunc,func,lnsrch
```

Given a starting point `p(1:n)` that is a vector of length `n`, the Broyden-Fletcher-Goldfarb-Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence requirement on zeroing the gradient is input as `gtol`. Returned quantities are `p(1:n)` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `lnsrch` is called to perform approximate line minimizations. Parameters: `NMAX` is the maximum anticipated value of `n`; `ITMAX` is the maximum allowed number of iterations; `STPMX` is the scaled maximum step length allowed in line searches; `TOLX` is the convergence criterion on x values.

```
      INTEGER i,its,j
      LOGICAL check
      REAL den,fac,fad,fae,fp,stpmax,sum,sumdg,sumxi,temp,test,
     *     dg(NMAX),g(NMAX),hdg(NMAX),hessin(NMAX,NMAX),
     *     pnew(NMAX),xi(NMAX)
      fp=func(p)                        Calculate starting function value and gradient,
      call dfunc(p,g)
      sum=0.
      do 12 i=1,n                       and initialize the inverse Hessian to the unit matrix.
        do 11 j=1,n
          hessin(i,j)=0.
        enddo 11
        hessin(i,i)=1.
        xi(i)=-g(i)                     Initial line direction.
        sum=sum+p(i)**2
      enddo 12
      stpmax=STPMX*max(sqrt(sum),float(n))
      do 27 its=1,ITMAX                 Main loop over the iterations.
        iter=its
        call lnsrch(n,p,fp,g,xi,pnew,fret,stpmax,check,func)
```

The new function evaluation occurs in `lnsrch`; save the function value in `fp` for the next line search. It is usually safe to ignore the value of `check`.

```
        fp=fret
        do 13 i=1,n
          xi(i)=pnew(i)-p(i)            Update the line direction,
          p(i)=pnew(i)                  and the current point.
        enddo 13
        test=0.                         Test for convergence on Δx.
        do 14 i=1,n
          temp=abs(xi(i))/max(abs(p(i)),1.)
          if(temp.gt.test)test=temp
        enddo 14
        if(test.lt.TOLX)return
        do 15 i=1,n                     Save the old gradient,
          dg(i)=g(i)
        enddo 15
        call dfunc(p,g)                 and get the new gradient.
        test=0.                         Test for convergence on zero gradient.
        den=max(fret,1.)
        do 16 i=1,n
          temp=abs(g(i))*max(abs(p(i)),1.)/den
          if(temp.gt.test)test=temp
        enddo 16
        if(test.lt.gtol)return
        do 17 i=1,n                     Compute difference of gradients,
          dg(i)=g(i)-dg(i)
        enddo 17
        do 19 i=1,n                     and difference times current matrix.
          hdg(i)=0.
```

```
        do 18 j=1,n
            hdg(i)=hdg(i)+hessin(i,j)*dg(j)
        enddo 18
    enddo 19
    fac=0.                          Calculate dot products for the denominators.
    fae=0.
    sumdg=0.
    sumxi=0.
    do 21 i=1,n
        fac=fac+dg(i)*xi(i)
        fae=fae+dg(i)*hdg(i)
        sumdg=sumdg+dg(i)**2
        sumxi=sumxi+xi(i)**2
    enddo 21
    if(fac.gt.sqrt(EPS*sumdg*sumxi))then  Skip update if fac not sufficiently positive.
        fac=1./fac
        fad=1./fae
        do 22 i=1,n                 The vector that makes BFGS different from DFP:
            dg(i)=fac*xi(i)-fad*hdg(i)
        enddo 22
        do 24 i=1,n                 The BFGS updating formula:
            do 23 j=i,n
                hessin(i,j)=hessin(i,j)+fac*xi(i)*xi(j)
*                -fad*hdg(i)*hdg(j)+fae*dg(i)*dg(j)
                hessin(j,i)=hessin(i,j)
            enddo 23
        enddo 24
    endif
    do 26 i=1,n                     Now calculate the next direction to go,
        xi(i)=0.
        do 25 j=1,n
            xi(i)=xi(i)-hessin(i,j)*g(j)
        enddo 25
    enddo 26
enddo 27                            and go back for another iteration.
pause 'too many iterations in dfpmin'
return
END
```

Quasi-Newton methods like `dfpmin` work well with the approximate line minimization done by `lnsrch`. The routines `powell` (§10.5) and `frprmn` (§10.6), however, need more accurate line minimization, which is carried out by the routine `linmin`.

Advanced Implementations of Variable Metric Methods

Although rare, it can conceivably happen that roundoff errors cause the matrix $\mathbf{H}_i$ to become nearly singular or non-positive-definite. This can be serious, because the supposed search directions might then not lead downhill, and because nearly singular $\mathbf{H}_i$'s tend to give subsequent $\mathbf{H}_i$'s that are also nearly singular.

There is a simple fix for this rare problem, the same as was mentioned in §10.4: In case of any doubt, you should *restart* the algorithm at the claimed minimum point, and see if it goes anywhere. Simple, but not very elegant. Modern implementations of variable metric methods deal with the problem in a more sophisticated way.

Instead of building up an approximation to $\mathbf{A}^{-1}$, it is possible to build up an approximation of $\mathbf{A}$ itself. Then, instead of calculating the left-hand side of (10.7.4) directly, one solves the set of linear equations

$$\mathbf{A} \cdot (\mathbf{x}_m - \mathbf{x}_i) = -\nabla f(\mathbf{x}_i) \tag{10.7.11}$$

At first glance this seems like a bad idea, since solving (10.7.11) is a process of order N^3 — and anyway, how does this help the roundoff problem? The trick is not to store $\mathbf{A}$ but

rather a triangular decomposition of **A**, its *Cholesky decomposition* (cf. §2.9). The updating formula used for the Cholesky decomposition of **A** is of order N^2 and can be arranged to guarantee that the matrix remains positive definite and nonsingular, even in the presence of finite roundoff. This method is due to Gill and Murray [1,2].

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1, §§3–6 (by K. W. Brodlie). [2]

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), pp. 56ff. [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 467–468.

10.8 Linear Programming and the Simplex Method

The subject of *linear programming*, sometimes called *linear optimization*, concerns itself with the following problem: For N independent variables $x_1, \ldots, x_N$, *maximize* the function

$$z = a_{01}x_1 + a_{02}x_2 + \cdots + a_{0N}x_N \tag{10.8.1}$$

subject to the primary constraints

$$x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots \quad x_N \geq 0 \tag{10.8.2}$$

and simultaneously subject to $M = m_1 + m_2 + m_3$ additional constraints, m_1 of them of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{iN}x_N \leq b_i \qquad (b_i \geq 0) \qquad i = 1, \ldots, m_1 \tag{10.8.3}$$

m_2 of them of the form

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jN}x_N \geq b_j \geq 0 \qquad j = m_1 + 1, \ldots, m_1 + m_2 \tag{10.8.4}$$

and m_3 of them of the form

$$\begin{aligned} a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kN}x_N &= b_k \geq 0 \\ k &= m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 \end{aligned} \tag{10.8.5}$$

The various a_{ij}'s can have either sign, or be zero. The fact that the b's must all be nonnegative (as indicated by the final inequality in the above three equations) is a matter of convention only, since you can multiply any contrary inequality by -1. There is no particular significance in the number of constraints M being less than, equal to, or greater than the number of unknowns N.

rather a triangular decomposition of **A**, its *Cholesky decomposition* (cf. §2.9). The updating formula used for the Cholesky decomposition of **A** is of order N^2 and can be arranged to guarantee that the matrix remains positive definite and nonsingular, even in the presence of finite roundoff. This method is due to Gill and Murray [1,2].

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1, §§3–6 (by K. W. Brodlie). [2]

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), pp. 56ff. [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 467–468.

10.8 Linear Programming and the Simplex Method

The subject of *linear programming*, sometimes called *linear optimization*, concerns itself with the following problem: For N independent variables $x_1, \ldots, x_N$, *maximize* the function

$$z = a_{01}x_1 + a_{02}x_2 + \cdots + a_{0N}x_N \tag{10.8.1}$$

subject to the primary constraints

$$x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots \quad x_N \geq 0 \tag{10.8.2}$$

and simultaneously subject to $M = m_1 + m_2 + m_3$ additional constraints, m_1 of them of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{iN}x_N \leq b_i \qquad (b_i \geq 0) \qquad i = 1, \ldots, m_1 \tag{10.8.3}$$

m_2 of them of the form

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jN}x_N \geq b_j \geq 0 \qquad j = m_1 + 1, \ldots, m_1 + m_2 \tag{10.8.4}$$

and m_3 of them of the form

$$\begin{aligned} a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kN}x_N &= b_k \geq 0 \\ k &= m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 \end{aligned} \tag{10.8.5}$$

The various a_{ij}'s can have either sign, or be zero. The fact that the b's must all be nonnegative (as indicated by the final inequality in the above three equations) is a matter of convention only, since you can multiply any contrary inequality by -1. There is no particular significance in the number of constraints M being less than, equal to, or greater than the number of unknowns N.

A set of values $x_1 \ldots x_N$ that satisfies the constraints (10.8.2)–(10.8.5) is called a *feasible vector*. The function that we are trying to maximize is called the *objective function*. The feasible vector that maximizes the objective function is called the *optimal feasible vector*. An optimal feasible vector can fail to exist for two distinct reasons: (i) there are *no* feasible vectors, i.e., the given constraints are incompatible, or (ii) there is no maximum, i.e., there is a direction in N space where one or more of the variables can be taken to infinity while still satisfying the constraints, giving an unbounded value for the objective function.

As you see, the subject of linear programming is surrounded by notational and terminological thickets. Both of these thorny defenses are lovingly cultivated by a coterie of stern acolytes who have devoted themselves to the field. Actually, the basic ideas of linear programming are quite simple. Avoiding the shrubbery, we want to teach you the basics by means of a couple of specific examples; it should then be quite obvious how to generalize.

Why is linear programming so important? (i) Because "nonnegativity" is the usual constraint on any variable x_i that represents the tangible amount of some physical commodity, like guns, butter, dollars, units of vitamin E, food calories, kilowatt hours, mass, etc. Hence equation (10.8.2). (ii) Because one is often interested in additive (linear) limitations or bounds imposed by man or nature: minimum nutritional requirement, maximum affordable cost, maximum on available labor or capital, minimum tolerable level of voter approval, etc. Hence equations (10.8.3)–(10.8.5). (iii) Because the function that one wants to optimize may be linear, or else may at least be approximated by a linear function — since that is the problem that linear programming *can* solve. Hence equation (10.8.1). For a short, semipopular survey of linear programming applications, see Bland [1].

Here is a specific example of a problem in linear programming, which has $N = 4$, $m_1 = 2$, $m_2 = m_3 = 1$, hence $M = 4$:

$$\text{Maximize} \quad z = x_1 + x_2 + 3x_3 - \tfrac{1}{2}x_4 \tag{10.8.6}$$

with all the x's nonnegative and also with

$$\begin{aligned} x_1 + 2x_3 &\le 740 \\ 2x_2 - 7x_4 &\le 0 \\ x_2 - x_3 + 2x_4 &\ge \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.7}$$

The answer turns out to be (to 2 decimals) $x_1 = 0$, $x_2 = 3.33$, $x_3 = 4.73$, $x_4 = 0.95$. In the rest of this section we will learn how this answer is obtained. Figure 10.8.1 summarizes some of the terminology thus far.

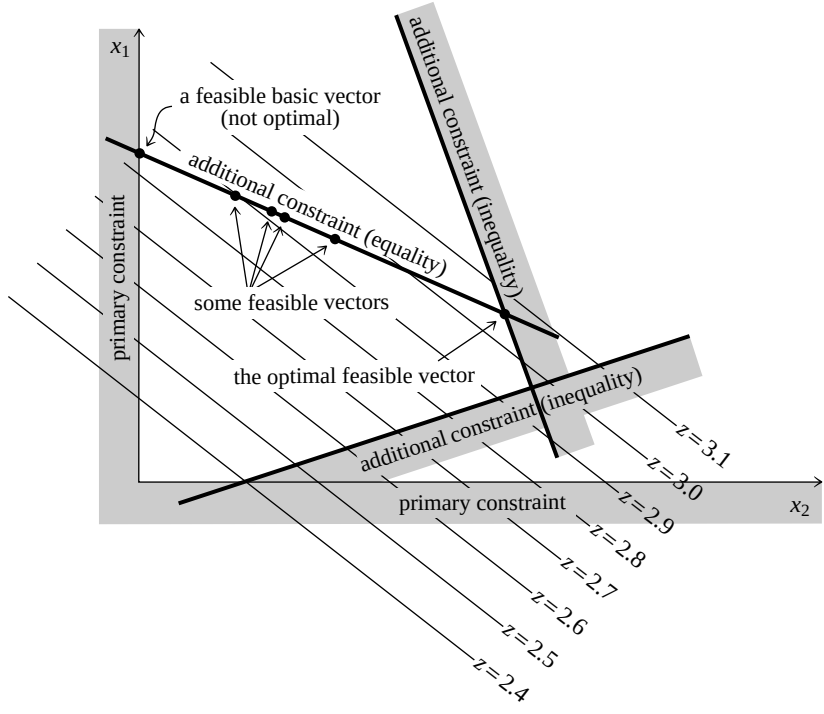


Figure 10.8.1. Basic concepts of linear programming. The case of only two independent variables, x_1, x_2, is shown. The linear function z, to be maximized, is represented by its contour lines. Primary constraints require x_1 and x_2 to be positive. Additional constraints may restrict the solution to regions (inequality constraints) or to surfaces of lower dimensionality (equality constraints). Feasible vectors satisfy all constraints. Feasible basic vectors also lie on the boundary of the allowed region. The simplex method steps among feasible basic vectors until the optimal feasible vector is found.

Fundamental Theorem of Linear Optimization

Imagine that we start with a full N-dimensional space of candidate vectors. Then (in mind's eye, at least) we carve away the regions that are eliminated in turn by each imposed constraint. Since the constraints are linear, every boundary introduced by this process is a plane, or rather hyperplane. Equality constraints of the form (10.8.5) force the feasible region onto hyperplanes of smaller dimension, while inequalities simply divide the then-feasible region into allowed and nonallowed pieces.

When all the constraints are imposed, either we are left with some feasible region or else there are no feasible vectors. Since the feasible region is bounded by hyperplanes, it is geometrically a kind of convex polyhedron or simplex (cf. §10.4). If there is a feasible region, can the optimal feasible vector be somewhere in its interior, away from the boundaries? No, because the objective function is linear. This means that it always has a nonzero vector gradient. This, in turn, means that we could always increase the objective function by running up the gradient until we hit a boundary wall.

The boundary of any geometrical region has one less dimension than its interior. Therefore, we can now run up the gradient projected into the boundary wall until we

reach an edge of that wall. We can then run up that edge, and so on, down through whatever number of dimensions, until we finally arrive at a point, a *vertex* of the original simplex. Since this point has all N of its coordinates defined, it must be the solution of N simultaneous *equalities* drawn from the original set of equalities and inequalities (10.8.2)–(10.8.5).

Points that are feasible vectors and that satisfy N of the original constraints as equalities, are termed *feasible basic vectors*. If $N > M$, then a feasible basic vector has *at least* $N - M$ of its components equal to zero, since at least that many of the constraints (10.8.2) will be needed to make up the total of N. Put the other way, *at most* M components of a feasible basic vector are nonzero. In the example (10.8.6)–(10.8.7), you can check that the solution as given satisfies as equalities the last three constraints of (10.8.7) and the constraint $x_1 \geq 0$, for the required total of 4.

Put together the two preceding paragraphs and you have the *Fundamental Theorem of Linear Optimization*: If an optimal feasible vector exists, then there is a feasible basic vector that is optimal. (Didn't we warn you about the terminological thicket?)

The importance of the fundamental theorem is that it reduces the optimization problem to a "combinatorial" problem, that of determining which N constraints (out of the $M + N$ constraints in 10.8.2–10.8.5) should be satisfied by the optimal feasible vector. We have only to keep trying different combinations, and computing the objective function for each trial, until we find the best.

Doing this blindly would take halfway to forever. The *simplex method*, first published by Dantzig in 1948 (see [2]), is a way of organizing the procedure so that (i) a series of combinations is tried for which the objective function increases at each step, and (ii) the optimal feasible vector is reached after a number of iterations that is almost always no larger than of order M or N, whichever is larger. An interesting mathematical sidelight is that this second property, although known empirically ever since the simplex method was devised, was not proved to be true until the 1982 work of Stephen Smale. (For a contemporary account, see [3].)

Simplex Method for a Restricted Normal Form

A linear programming problem is said to be in *normal form* if it has no constraints in the form (10.8.3) or (10.8.4), but rather only equality constraints of the form (10.8.5) and nonnegativity constraints of the form (10.8.2).

For our purposes it will be useful to consider an even more restricted set of cases, with this additional property: Each equality constraint of the form (10.8.5) must have at least one variable that has a positive coefficient and *that appears uniquely in that one constraint only*. We can then choose one such variable in each constraint equation, and solve that constraint equation for it. The variables thus chosen are called *left-hand variables* or *basic variables*, and there are exactly M ($= m_3$) of them. The remaining $N - M$ variables are called *right-hand variables* or *nonbasic variables*. Obviously this *restricted normal form* can be achieved only in the case $M \leq N$, so that is the case that we will consider.

You may be thinking that our restricted normal form is so specialized that it is unlikely to include the linear programming problem that you wish to solve. Not at all! We will presently show how *any* linear programming problem can be

transformed into restricted normal form. Therefore bear with us and learn how to apply the simplex method to a restricted normal form.

Here is an example of a problem in restricted normal form:

$$\text{Maximize} \quad z = 2x_2 - 4x_3 \tag{10.8.8}$$

with x_1, x_2, x_3, and x_4 all nonnegative and also with

$$\begin{aligned} x_1 &= 2 - 6x_2 + x_3 \\ x_4 &= 8 + 3x_2 - 4x_3 \end{aligned} \tag{10.8.9}$$

This example has $N = 4$, $M = 2$; the left-hand variables are x_1 and x_4; the right-hand variables are x_2 and x_3. The objective function (10.8.8) is written so as to depend only on right-hand variables; note, however, that this is not an actual restriction on objective functions in restricted normal form, since any left-hand variables appearing in the objective function could be eliminated algebraically by use of (10.8.9) or its analogs.

For any problem in restricted normal form, we can instantly read off a feasible basic vector (although not necessarily the *optimal* feasible basic vector). Simply set all right-hand variables equal to zero, and equation (10.8.9) then gives the values of the left-hand variables for which the constraints are satisfied. The idea of the simplex method is to proceed by a series of exchanges. In each exchange, a right-hand variable and a left-hand variable change places. At each stage we maintain a problem in restricted normal form that is equivalent to the original problem.

It is notationally convenient to record the information content of equations (10.8.8) and (10.8.9) in a so-called *tableau*, as follows:

		x_2	x_3
z	0	2	-4
x_1	2	-6	1
x_4	8	3	-4

(10.8.10)

You should study (10.8.10) to be sure that you understand where each entry comes from, and how to translate back and forth between the tableau and equation formats of a problem in restricted normal form.

The first step in the simplex method is to examine the top row of the tableau, which we will call the "z-row." Look at the entries in columns labeled by right-hand variables (we will call these "right-columns"). We want to imagine in turn the effect of increasing each right-hand variable from its present value of zero, while leaving all the other right-hand variables at zero. Will the objective function increase or decrease? The answer is given by the sign of the entry in the z-row. Since we want to increase the objective function, only right columns having positive z-row entries are of interest. In (10.8.10) there is only one such column, whose z-row entry is 2.

The second step is to examine the column entries below each z-row entry that was selected by step one. We want to ask how much we can increase the right-hand variable before one of the left-hand variables is driven negative, which is not allowed. If the tableau element at the intersection of the right-hand column and

the left-hand variable's row is positive, then it poses no restriction: the corresponding left-hand variable will just be driven more and more positive. If *all* the entries in any right-hand column are positive, then there is no bound on the objective function and (having said so) we are done with the problem.

If one or more entries below a positive z-row entry are negative, then we have to figure out which such entry first limits the increase of that column's right-hand variable. Evidently the limiting increase is given by dividing the element in the right-hand column (which is called the *pivot element*) into the element in the "constant column" (leftmost column) of the pivot element's row. A value that is small in magnitude is most restrictive. The increase in the objective function for this choice of pivot element is then that value multiplied by the z-row entry of that column. We repeat this procedure on all possible right-hand columns to find the pivot element with the largest such increase. That completes our "choice of a pivot element."

In the above example, the only positive z-row entry is 2. There is only one negative entry below it, namely -6, so this is the pivot element. Its constant-column entry is 2. This pivot will therefore allow x_2 to be increased by $2 \div |6|$, which results in an increase of the objective function by an amount $(2 \times 2) \div |6|$.

The third step is to *do* the increase of the selected right-hand variable, thus making it a left-hand variable; and simultaneously to modify the left-hand variables, reducing the pivot-row element to zero and thus making it a right-hand variable. For our above example let's do this first by hand: We begin by solving the pivot-row equation for the new left-hand variable x_2 in favor of the old one x_1, namely

$$x_1 = 2 - 6x_2 + x_3 \qquad \rightarrow \qquad x_2 = \tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3 \tag{10.8.11}$$

We then substitute this into the old z-row,

$$z = 2x_2 - 4x_3 = 2\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = \tfrac{2}{3} - \tfrac{1}{3}x_1 - \tfrac{11}{3}x_3 \tag{10.8.12}$$

and into all other left-variable rows, in this case only x_4,

$$x_4 = 8 + 3\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = 9 - \tfrac{1}{2}x_1 - \tfrac{7}{2}x_3 \tag{10.8.13}$$

Equations (10.8.11)–(10.8.13) form the new tableau

		x_1	x_3
z	$\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{11}{3}$
x_2	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$
x_4	9	$-\frac{1}{2}$	$-\frac{7}{2}$

(10.8.14)

The fourth step is to go back and repeat the first step, looking for another possible increase of the objective function. We do this as many times as possible, that is, until all the right-hand entries in the z-row are negative, signaling that no further increase is possible. In the present example, this already occurs in (10.8.14), so we are done.

The answer can now be read from the constant column of the final tableau. In (10.8.14) we see that the objective function is maximized to a value of $2/3$ for the solution vector $x_2 = 1/3$, $x_4 = 9$, $x_1 = x_3 = 0$.

Now look back over the procedure that led from (10.8.10) to (10.8.14). You will find that it could be summarized entirely in tableau format as a series of prescribed elementary matrix operations:

- Locate the pivot element and save it.
- Save the whole pivot column.
- Replace each row, except the pivot row, by that linear combination of itself and the pivot row which makes its pivot-column entry zero.
- Divide the pivot row by the negative of the pivot.
- Replace the pivot element by the reciprocal of its saved value.
- Replace the rest of the pivot column by its saved values divided by the saved pivot element.

This is the sequence of operations actually performed by a linear programming routine, such as the one that we will presently give.

You should now be able to solve almost any linear programming problem that starts in restricted normal form. The only special case that might stump you is if an entry in the constant column turns out to be zero at some stage, so that a left-hand variable is zero at the same time as all the right-hand variables are zero. This is called a *degenerate feasible vector*. To proceed, you may need to exchange the degenerate left-hand variable for one of the right-hand variables, perhaps even making several such exchanges.

Writing the General Problem in Restricted Normal Form

Here is a pleasant surprise. There exist a couple of clever tricks that render trivial the task of translating a general linear programming problem into restricted normal form!

First, we need to get rid of the inequalities of the form (10.8.3) or (10.8.4), for example, the first three constraints in (10.8.7). We do this by adding to the problem so-called *slack variables* which, when their nonnegativity is required, convert the inequalities to equalities. We will denote slack variables as y_i. There will be $m_1 + m_2$ of them. Once they are introduced, you treat them on an equal footing with the original variables x_i; then, at the very end, you simply ignore them.

For example, introducing slack variables leaves (10.8.6) unchanged but turns (10.8.7) into

$$\begin{aligned} x_1 + 2x_3 + y_1 &= 740 \\ 2x_2 - 7x_4 + y_2 &= 0 \\ x_2 - x_3 + 2x_4 - y_3 &= \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.15}$$

(Notice how the sign of the coefficient of the slack variable is determined by which sense of inequality it is replacing.)

Second, we need to insure that there is a set of M left-hand vectors, so that we can set up a starting tableau in restricted normal form. (In other words, we need to find a "feasible basic starting vector.") The trick is again to invent new variables! There are M of these, and they are called *artificial variables*; we denote them by z_i. You put exactly one artificial variable into each constraint equation on the following

model for the example (10.8.15):

$$\begin{aligned} z_1 &= 740 - x_1 - 2x_3 - y_1 \\ z_2 &= -2x_2 + 7x_4 - y_2 \\ z_3 &= \tfrac{1}{2} - x_2 + x_3 - 2x_4 + y_3 \\ z_4 &= 9 - x_1 - x_2 - x_3 - x_4 \end{aligned} \tag{10.8.16}$$

Our example is now in restricted normal form.

Now you may object that (10.8.16) is not the same problem as (10.8.15) or (10.8.7) *unless all the z_i's are zero*. Right you are! There is some subtlety here! We must proceed to solve our problem in two phases. First phase: We replace our objective function (10.8.6) by a so-called *auxiliary objective function*

$$z' \equiv -z_1 - z_2 - z_3 - z_4 = -(749\tfrac{1}{2} - 2x_1 - 4x_2 - 2x_3 + 4x_4 - y_1 - y_2 + y_3) \tag{10.8.17}$$

(where the last equality follows from using 10.8.16). We now perform the simplex method on the auxiliary objective function (10.8.17) with the constraints (10.8.16). Obviously the auxiliary objective function will be maximized for nonnegative z_i's if all the z_i's are zero. We therefore expect the simplex method in this first phase to produce a set of left-hand variables drawn from the x_i's and y_i's only, with all the z_i's being right-hand variables. Aha! We then cross out the z_i's, leaving a problem involving only x_i's and y_i's in restricted normal form. In other words, the first phase produces an initial feasible basic vector. Second phase: Solve the problem produced by the first phase, using the original objective function, not the auxiliary.

And what if the first phase *doesn't* produce zero values for all the z_i's? That signals that there is *no* initial feasible basic vector, i.e., that the constraints given to us are inconsistent among themselves. Report that fact, and you are done.

Here is how to translate into tableau format the information needed for both the first and second phases of the overall method. As before, the underlying problem to be solved is as posed in equations (10.8.6)–(10.8.7).

		x_1	x_2	x_3	x_4	y_1	y_2	y_3
z	0	1	1	3	$-\frac{1}{2}$	0	0	0
z_1	740	-1	0	-2	0	-1	0	0
z_2	0	0	-2	0	7	0	-1	0
z_3	$\frac{1}{2}$	0	-1	1	-2	0	0	1
z_4	9	-1	-1	-1	-1	0	0	0
z'	$-749\frac{1}{2}$	2	4	2	-4	1	1	-1

(10.8.18)

This is not as daunting as it may, at first sight, appear. The table entries inside the box of double lines are no more than the coefficients of the original problem (10.8.6)–(10.8.7) organized into a tabular form. In fact, these entries, along with

the values of N, M, m_1, m_2, and m_3, are the only input that is needed by the simplex method routine below. The columns under the slack variables y_i simply record whether each of the M constraints is of the form $\leq$, $\geq$, or $=$; this is redundant information with the values m_1, m_2, m_3, as long as we are sure to enter the rows of the tableau in the correct respective order. The coefficients of the auxiliary objective function (bottom row) are just the negatives of the column sums of the rows above, so these are easily calculated automatically.

The output from a simplex routine will be (i) a flag telling whether a finite solution, no solution, or an unbounded solution was found, and (ii) an updated tableau. The output tableau that derives from (10.8.18), given to two significant figures, is

		x_1	y_2	y_3	$\cdots$
z	17.03	$-.95$	$-.05$	-1.05	$\cdots$
x_2	3.33	$-.35$	$-.15$	.35	$\cdots$
x_3	4.73	$-.55$	.05	$-.45$	$\cdots$
x_4	.95	$-.10$	.10	.10	$\cdots$
y_1	730.55	.10	$-.10$	.90	$\cdots$

(10.8.19)

A little counting of the x_i's and y_i's will convince you that there are $M + 1$ rows (including the z-row) in both the input and the output tableaux, but that only $N + 1 - m_3$ columns of the output tableau (including the constant column) contain any useful information, the other columns belonging to now-discarded artificial variables. In the output, the first numerical column contains the solution vector, along with the maximum value of the objective function. Where a slack variable (y_i) appears on the left, the corresponding value is the amount by which its inequality is safely satisfied. Variables that are not left-hand variables in the output tableau have zero values. Slack variables with zero values represent constraints that are satisfied as equalities.

Routine Implementing the Simplex Method

The following routine is based algorithmically on the implementation of Kuenzi, Tzschach, and Zehnder [4]. Aside from input values of M, N, m_1, m_2, m_3, the principal input to the routine is a two-dimensional array `a` containing the portion of the tableau (10.8.18) that is contained between the double lines. This input occupies the first $M + 1$ rows and $N + 1$ columns of `a`. Note, however, that reference is made internally to row $M + 2$ of `a` (used for the auxiliary objective function, just as in 10.8.18). Therefore the physical dimensions of `a`,

```
REAL a(MP,NP)
```
(10.8.20)

must have `NP`$\geq N + 1$ and `MP`$\geq M + 2$.You will suffer endless agonies if you fail to understand this simple point. Also do not neglect to order the rows of `a` in the same order as equations (10.8.1), (10.8.3), (10.8.4), and (10.8.5), that is, objective function, $\leq$-constraints, $\geq$-constraints, $=$-constraints.

On output, the tableau a is indexed by two returned arrays of integers. iposv(j) contains, for j= $1 \ldots M$, the number i whose original variable x_i is now represented by row j+1 of a. These are thus the left-hand variables in the solution. (The first row of a is of course the z-row.) A value $i > N$ indicates that the variable is a y_i rather than an x_i, $x_{N+j} \equiv y_j$. Likewise, izrov(j) contains, for j= $1 \ldots N$, the number i whose original variable x_i is now a right-hand variable, represented by column j+1 of a. These variables are all zero in the solution. The meaning of $i > N$ is the same as above, except that $i > N + m_1 + m_2$ denotes an artificial or slack variable which was used only internally and should now be entirely ignored.

The flag icase is returned as zero if a finite solution is found, $+1$ if the objective function is unbounded, -1 if no solution satisfies the given constraints.

The routine treats the case of degenerate feasible vectors, so don't worry about them. You may also wish to admire the fact that the routine does not require storage for the columns of the tableau (10.8.18) that are to the right of the double line; it keeps track of slack variables by more efficient bookkeeping.

Please note that, as given, the routine is only "semi-sophisticated" in its tests for convergence. While the routine properly implements tests for inequality with zero as tests against some small parameter EPS, it does not adjust this parameter to reflect the scale of the input data. This is adequate for many problems, where the input data do not differ from unity by too many orders of magnitude. If, however, you encounter endless cycling, then you should modify EPS in the routines simplx and simp2. Permuting your variables can also help. Finally, consult [5].

```
      SUBROUTINE simplx(a,m,n,mp,np,m1,m2,m3,icase,izrov,iposv)
      INTEGER icase,m,m1,m2,m3,mp,n,np,iposv(m),izrov(n),MMAX,NMAX
      REAL a(mp,np),EPS
      PARAMETER (MMAX=100,NMAX=100,EPS=1.e-6)
C     USES simp1,simp2,simp3
```

Simplex method for linear programming. Input parameters a, m, n, mp, np, m1, m2, and m3, and output parameters a, icase, izrov, and iposv are described above.
Parameters: MMAX is the maximum number of constraints expected; NMAX is the maximum number of variables expected; EPS is the absolute precision, which should be adjusted to the scale of your variables.

```
      INTEGER i,ip,is,k,kh,kp,nl1,l1(NMAX),l3(MMAX)
      REAL bmax,q1
      if(m.ne.m1+m2+m3)pause 'bad input constraint counts in simplx'
      nl1=n
      do 11 k=1,n
        l1(k)=k                    Initialize index list of columns admissible for exchange.
        izrov(k)=k                 Initially make all variables right-hand.
      enddo 11
      do 12 i=1,m
        if(a(i+1,1).lt.0.)pause 'bad input tableau in simplx'   Constants b_i must be non-
        iposv(i)=n+i                                                negative.
```

Initial left-hand variables. m1 type constraints are represented by having their slack variable initially left-hand, with no artificial variable. m2 type constraints have their slack variable initially left-hand, with a minus sign, and their artificial variable handled implicitly during their first exchange. m3 type constraints have their artificial variable initially left-hand.

```
      enddo 12
      if(m2+m3.eq.0)goto 30        The origin is a feasible starting solution. Go to phase two.
      do 13 i=1,m2                 Initialize list of m2 constraints whose slack variables have never
        l3(i)=1                        been exchanged out of the initial basis.
      enddo 13
      do 15 k=1,n+1                Compute the auxiliary objective function.
        q1=0.
        do 14 i=m1+1,m
```

```
            q1=q1+a(i+1,k)
        enddo 14
        a(m+2,k)=-q1
    enddo 15
10  call simp1(a,mp,np,m+1,l1,nl1,0,kp,bmax)        Find max. coeff. of auxiliary objec-
    if(bmax.le.EPS.and.a(m+2,1).lt.-EPS)then            tive fn.
        icase=-1                                 Auxiliary objective function is still negative and can't be im-
        return                                       proved, hence no feasible solution exists.
    else if(bmax.le.EPS.and.a(m+2,1).le.EPS)then
      Auxiliary objective function is zero and can't be improved; we have a feasible starting vec-
      tor. Clean out the artificial variables corresponding to any remaining equality constraints by
      goto 1's and then move on to phase two by goto 30.
        do 16 ip=m1+m2+1,m
            if(iposv(ip).eq.ip+n)then            Found an artificial variable for an equality
                call simp1(a,mp,np,ip,l1,nl1,1,kp,bmax)      constraint.
                if(bmax.gt.EPS)goto 1            Exchange with column corresponding to max-
            endif                                    imum pivot element in row.
        enddo 16
        do 18 i=m1+1,m1+m2                       Change sign of row for any m2 constraints
            if(l3(i-m1).eq.1)then                    still present from the initial basis.
                do 17 k=1,n+1
                    a(i+1,k)=-a(i+1,k)
                enddo 17
            endif
        enddo 18
        goto 30                                  Go to phase two.
    endif
    call simp2(a,m,n,mp,np,ip,kp)                Locate a pivot element (phase one).
    if(ip.eq.0)then                              Maximum of auxiliary objective function is
        icase=-1                                     unbounded, so no feasible solution ex-
        return                                       ists.
    endif
1   call simp3(a,mp,np,m+1,n,ip,kp)
      Exchange a left- and a right-hand variable (phase one), then update lists.
    if(iposv(ip).ge.n+m1+m2+1)then               Exchanged out an artificial variable for an
        do 19 k=1,nl1                                equality constraint. Make sure it stays
            if(l1(k).eq.kp)goto 2                    out by removing it from the l1 list.
        enddo 19
2       nl1=nl1-1
        do 21 is=k,nl1
            l1(is)=l1(is+1)
        enddo 21
    else
        kh=iposv(ip)-m1-n
        if(kh.ge.1)then                          Exchanged out an m2 type constraint.
            if(l3(kh).ne.0)then                  If it's the first time, correct the pivot col-
                l3(kh)=0                             umn for the minus sign and the implicit
                a(m+2,kp+1)=a(m+2,kp+1)+1.           artificial variable.
                do 22 i=1,m+2
                    a(i,kp+1)=-a(i,kp+1)
                enddo 22
            endif
        endif
    endif
    is=izrov(kp)                                 Update lists of left- and right-hand variables.
    izrov(kp)=iposv(ip)
    iposv(ip)=is
    goto 10                                      Still in phase one, go back to 10.
      End of phase one code for finding an initial feasible solution. Now, in phase two, optimize it.
30  call simp1(a,mp,np,0,l1,nl1,0,kp,bmax)          Test the z-row for doneness.
    if(bmax.le.EPS)then                      Done. Solution found. Return with the good news.
        icase=0
        return
    endif
```

```
      call simp2(a,m,n,mp,np,ip,kp)   Locate a pivot element (phase two).
      if(ip.eq.0)then                 Objective function is unbounded. Report and return.
        icase=1
        return
      endif
      call simp3(a,mp,np,m,n,ip,kp)   Exchange a left- and a right-hand variable (phase two),
      is=izrov(kp)                    update lists of left- and right-hand variables,
      izrov(kp)=iposv(ip)
      iposv(ip)=is
      goto 30                         and return for another iteration.
      END
```

The preceding routine makes use of the following utility subroutines.

```
      SUBROUTINE simp1(a,mp,np,mm,ll,nll,iabf,kp,bmax)
      INTEGER iabf,kp,mm,mp,nll,np,ll(np)
      REAL bmax,a(mp,np)
        Determines the maximum of those elements whose index is contained in the supplied list
        ll, either with or without taking the absolute value, as flagged by iabf.
      INTEGER k
      REAL test
      if(nll.le.0)then                No eligible columns.
        bmax=0.
      else
        kp=ll(1)
        bmax=a(mm+1,kp+1)
        do 11 k=2,nll
          if(iabf.eq.0)then
            test=a(mm+1,ll(k)+1)-bmax
          else
            test=abs(a(mm+1,ll(k)+1))-abs(bmax)
          endif
          if(test.gt.0.)then
            bmax=a(mm+1,ll(k)+1)
            kp=ll(k)
          endif
        enddo 11
      endif
      return
      END


      SUBROUTINE simp2(a,m,n,mp,np,ip,kp)
      INTEGER ip,kp,m,mp,n,np
      REAL a(mp,np),EPS
      PARAMETER (EPS=1.e-6)
        Locate a pivot element, taking degeneracy into account.
      INTEGER i,k
      REAL q,q0,q1,qp
      ip=0
      do 11 i=1,m
        if(a(i+1,kp+1).lt.-EPS)goto 1
      enddo 11
      return                                  No possible pivots. Return with message.
1     q1=-a(i+1,1)/a(i+1,kp+1)
      ip=i
      do 13 i=ip+1,m
        if(a(i+1,kp+1).lt.-EPS)then
          q=-a(i+1,1)/a(i+1,kp+1)
          if(q.lt.q1)then
            ip=i
            q1=q
          else if (q.eq.q1) then              We have a degeneracy.
```

```
            do 12 k=1,n
                qp=-a(ip+1,k+1)/a(ip+1,kp+1)
                q0=-a(i+1,k+1)/a(i+1,kp+1)
                if(q0.ne.qp)goto 2
            enddo 12
2           if(q0.lt.qp)ip=i
        endif
    endif
enddo 13
return
END


SUBROUTINE simp3(a,mp,np,i1,k1,ip,kp)
INTEGER i1,ip,k1,kp,mp,np
REAL a(mp,np)
    Matrix operations to exchange a left-hand and right-hand variable (see text).
INTEGER ii,kk
REAL piv
piv=1./a(ip+1,kp+1)
do 12 ii=1,i1+1
    if(ii-1.ne.ip)then
        a(ii,kp+1)=a(ii,kp+1)*piv
        do 11 kk=1,k1+1
            if(kk-1.ne.kp)then
                a(ii,kk)=a(ii,kk)-a(ip+1,kk)*a(ii,kp+1)
            endif
        enddo 11
    endif
enddo 12
do 13 kk=1,k1+1
    if(kk-1.ne.kp)a(ip+1,kk)=-a(ip+1,kk)*piv
enddo 13
a(ip+1,kp+1)=piv
return
END
```

Other Topics Briefly Mentioned

Every linear programming problem in normal form with N variables and M constraints has a corresponding *dual* problem with M variables and N constraints. The tableau of the dual problem is, in essence, the transpose of the tableau of the original (sometimes called *primal*) problem. It is possible to go from a solution of the dual to a solution of the primal. This can occasionally be computationally useful, but generally it is no big deal.

The *revised simplex method* is exactly equivalent to the simplex method in its choice of which left-hand and right-hand variables are exchanged. Its computational effort is not significantly less than that of the simplex method. It does differ in the organization of its storage, requiring only a matrix of size $M \times M$, rather than $M \times N$, in its intermediate stages. If you have a lot of constraints, and memory size is one of them, then you should look into it.

The *primal-dual algorithm* and the *composite simplex algorithm* are two different methods for avoiding the two phases of the usual simplex method: Progress is made simultaneously towards finding a feasible solution and finding an optimal solution. There seems to be no clearcut evidence that these methods are superior

to the usual method by any factor substantially larger than the "tender-loving-care factor" (which reflects the programming effort of the proponents).

Problems where the objective function and/or one or more of the constraints are replaced by expressions nonlinear in the variables are called *nonlinear programming problems*. The literature on such problems is vast, but outside our scope. The special case of quadratic expressions is called *quadratic programming*. Optimization problems where the variables take on only integer values are called *integer programming* problems, a special case of *discrete optimization* generally. The next section looks at a particular kind of discrete optimization problem.

CITED REFERENCES AND FURTHER READING:

Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp. 126–144. [1]

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]

Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), Chapters 7–8.

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders).

Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill).

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley).

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience).

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.10.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

10.9 Simulated Annealing Methods

The *method of simulated annealing* [1,2] is a technique that has attracted significant attention as suitable for optimization problems of large scale, especially ones where a desired global extremum is hidden among many, poorer, local extrema. For practical purposes, simulated annealing has effectively "solved" the famous *traveling salesman problem* of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn. (Other practical methods have also been found.) The method has also been used successfully for designing complex integrated circuits: The arrangement of several hundred thousand circuit elements on a tiny silicon substrate is optimized so as to minimize interference among their connecting wires [3,4]. Surprisingly, the implementation of the algorithm is relatively simple.

Notice that the two applications cited are both examples of *combinatorial minimization*. There is an objective function to be minimized, as usual; but the space over which that function is defined is not simply the N-dimensional space of N continuously variable parameters. Rather, it is a discrete, but very large, configuration

to the usual method by any factor substantially larger than the "tender-loving-care factor" (which reflects the programming effort of the proponents).

Problems where the objective function and/or one or more of the constraints are replaced by expressions nonlinear in the variables are called *nonlinear programming problems*. The literature on such problems is vast, but outside our scope. The special case of quadratic expressions is called *quadratic programming*. Optimization problems where the variables take on only integer values are called *integer programming* problems, a special case of *discrete optimization* generally. The next section looks at a particular kind of discrete optimization problem.

CITED REFERENCES AND FURTHER READING:

Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp. 126–144. [1]

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]

Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), Chapters 7–8.

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders).

Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill).

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley).

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience).

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.10.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

10.9 Simulated Annealing Methods

The *method of simulated annealing* [1,2] is a technique that has attracted significant attention as suitable for optimization problems of large scale, especially ones where a desired global extremum is hidden among many, poorer, local extrema. For practical purposes, simulated annealing has effectively "solved" the famous *traveling salesman problem* of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn. (Other practical methods have also been found.) The method has also been used successfully for designing complex integrated circuits: The arrangement of several hundred thousand circuit elements on a tiny silicon substrate is optimized so as to minimize interference among their connecting wires [3,4]. Surprisingly, the implementation of the algorithm is relatively simple.

Notice that the two applications cited are both examples of *combinatorial minimization*. There is an objective function to be minimized, as usual; but the space over which that function is defined is not simply the N-dimensional space of N continuously variable parameters. Rather, it is a discrete, but very large, configuration

space, like the set of possible orders of cities, or the set of possible allocations of silicon "real estate" blocks to circuit elements. The number of elements in the configuration space is factorially large, so that they cannot be explored exhaustively. Furthermore, since the set is discrete, we are deprived of any notion of "continuing downhill in a favorable direction." The concept of "direction" may not have any meaning in the configuration space.

Below, we will also discuss how to use simulated annealing methods for spaces with continuous control parameters, like those of §§10.4–10.7. This application is actually more complicated than the combinatorial one, since the familiar problem of "long, narrow valleys" again asserts itself. Simulated annealing, as we will see, tries "random" steps; but in a long, narrow valley, almost all random steps are uphill! Some additional finesse is therefore required.

At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize, or metals cool and anneal. At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is lost. The atoms are often able to line themselves up and form a pure crystal that is completely ordered over a distance up to billions of times the size of an individual atom in all directions. This crystal is the state of minimum energy for this system. The amazing fact is that, for slowly cooled systems, nature is able to find this minimum energy state. In fact, if a liquid metal is cooled quickly or "quenched," it does not reach this state but rather ends up in a polycrystalline or amorphous state having somewhat higher energy.

So the essence of the process is *slow* cooling, allowing ample time for redistribution of the atoms as they lose mobility. This is the technical definition of *annealing*, and it is essential for ensuring that a low energy state will be achieved.

Although the analogy is not perfect, there is a sense in which all of the minimization algorithms thus far in this chapter correspond to rapid cooling or quenching. In all cases, we have gone greedily for the quick, nearby solution: From the starting point, go immediately downhill as far as you can go. This, as often remarked above, leads to a local, but not necessarily a global, minimum. Nature's own minimization algorithm is based on quite a different procedure. The so-called Boltzmann probability distribution,

$$\text{Prob}\,(E) \sim \exp(-E/kT) \tag{10.9.1}$$

expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E. Even at low temperature, there is a chance, albeit very small, of a system being in a high energy state. Therefore, there is a corresponding chance for the system to get out of a local energy minimum in favor of finding a better, more global, one. The quantity k (Boltzmann's constant) is a constant of nature that relates temperature to energy. In other words, the system sometimes goes *uphill* as well as downhill; but the lower the temperature, the less likely is any significant uphill excursion.

In 1953, Metropolis and coworkers [5] first incorporated these kinds of principles into numerical calculations. Offered a succession of options, a simulated thermodynamic system was assumed to change its configuration from energy E_1 to energy E_2 with probability $p = \exp[-(E_2 - E_1)/kT]$. Notice that if $E_2 < E_1$, this probability is greater than unity; in such cases the change is arbitrarily assigned a probability $p = 1$, i.e., the system *always* took such an option. This general scheme,

of always taking a downhill step while *sometimes* taking an uphill step, has come to be known as the Metropolis algorithm.

To make use of the Metropolis algorithm for other than thermodynamic systems, one must provide the following elements:

1. A description of possible system configurations.

2. A generator of random changes in the configuration; these changes are the "options" presented to the system.

3. An objective function E (analog of energy) whose minimization is the goal of the procedure.

4. A control parameter T (analog of temperature) and an *annealing schedule* which tells how it is lowered from high to low values, e.g., after how many random changes in configuration is each downward step in T taken, and how large is that step. The meaning of "high" and "low" in this context, and the assignment of a schedule, may require physical insight and/or trial-and-error experiments.

Combinatorial Minimization: The Traveling Salesman

A concrete illustration is provided by the traveling salesman problem. The proverbial seller visits N cities with given positions (x_i, y_i), returning finally to his or her city of origin. Each city is to be visited only once, and the route is to be made as short as possible. This problem belongs to a class known as *NP-complete* problems, whose computation time for an *exact* solution increases with N as $\exp(\text{const.} \times N)$, becoming rapidly prohibitive in cost as N increases. The traveling salesman problem also belongs to a class of minimization problems for which the objective function E has many local minima. In practical cases, it is often enough to be able to choose from these a minimum which, even if not absolute, cannot be significantly improved upon. The annealing method manages to achieve this, while limiting its calculations to scale as a small power of N.

As a problem in simulated annealing, the traveling salesman problem is handled as follows:

1. *Configuration.* The cities are numbered $i = 1 \ldots N$ and each has coordinates (x_i, y_i). A configuration is a permutation of the number $1 \ldots N$, interpreted as the order in which the cities are visited.

2. *Rearrangements.* An efficient set of moves has been suggested by Lin [6]. The moves consist of two types: (a) A section of path is removed and then replaced with the same cities running in the opposite order; or (b) a section of path is removed and then replaced in between two cities on another, randomly chosen, part of the path.

3. *Objective Function.* In the simplest form of the problem, E is taken just as the total length of journey,

$$E = L \equiv \sum_{i=1}^{N} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{10.9.2}$$

with the convention that point $N + 1$ is identified with point 1. To illustrate the flexibility of the method, however, we can add the following additional wrinkle: Suppose that the salesman has an irrational fear of flying over the Mississippi River. In that case, we would assign each city a parameter μ_i, equal to $+1$ if it is east of the

Mississippi, -1 if it is west, and take the objective function to be

$$E = \sum_{i=1}^{N} \left[\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2 \right] \tag{10.9.3}$$

A penalty 4λ is thereby assigned to any river crossing. The algorithm now finds the shortest path that avoids crossings. The relative importance that it assigns to length of path versus river crossings is determined by our choice of λ. Figure 10.9.1 shows the results obtained. Clearly, this technique can be generalized to include many conflicting goals in the minimization.

4. *Annealing schedule.* This requires experimentation. We first generate some random rearrangements, and use them to determine the range of values of ΔE that will be encountered from move to move. Choosing a starting value for the parameter T which is considerably larger than the largest ΔE normally encountered, we proceed downward in multiplicative steps each amounting to a 10 percent decrease in T. We hold each new value of T constant for, say, $100N$ reconfigurations, or for $10N$ successful reconfigurations, whichever comes first. When efforts to reduce E further become sufficiently discouraging, we stop.

The following traveling salesman program, using the Metropolis algorithm, illustrates the main aspects of the simulated annealing technique for combinatorial problems.

```
      SUBROUTINE anneal(x,y,iorder,ncity)
      INTEGER ncity,iorder(ncity)
      REAL x(ncity),y(ncity)
C     USES irbit1,metrop,ran3,revcst,revers,trncst,trnspt
        This algorithm finds the shortest round-trip path to ncity cities whose coordinates are in
        the arrays x(1:ncity),y(1:ncity). The array iorder(1:ncity) specifies the order
        in which the cities are visited. On input, the elements of iorder may be set to any per-
        mutation of the numbers 1 to ncity. This routine will return the best alternative path
        it can find.
      INTEGER i,i1,i2,idec,idum,iseed,j,k,nlimit,nn,nover,nsucc,n(6),
     *     irbit1
      REAL de,path,t,tfactr,ran3,alen,x1,x2,y1,y2
      LOGICAL ans
      alen(x1,x2,y1,y2)=sqrt((x2-x1)**2+(y2-y1)**2)
      nover=100*ncity           Maximum number of paths tried at any temperature.
      nlimit=10*ncity           Maximum number of successful path changes before continuing.
      tfactr=0.9                Annealing schedule: t is reduced by this factor on each step.
      path=0.0
      t=0.5
      do 11 i=1,ncity-1         Calculate initial path length.
        i1=iorder(i)
        i2=iorder(i+1)
        path=path+alen(x(i1),x(i2),y(i1),y(i2))
      enddo 11
      i1=iorder(ncity)          Close the loop by tying path ends together.
      i2=iorder(1)
      path=path+alen(x(i1),x(i2),y(i1),y(i2))
      idum=-1
      iseed=111
      do 13 j=1,100             Try up to 100 temperature steps.
        nsucc=0
        do 12 k=1,nover
1         n(1)=1+int(ncity*ran3(idum))          Choose beginning of segment ..
          n(2)=1+int((ncity-1)*ran3(idum))      ..and end of segment.
          if (n(2).ge.n(1)) n(2)=n(2)+1
```

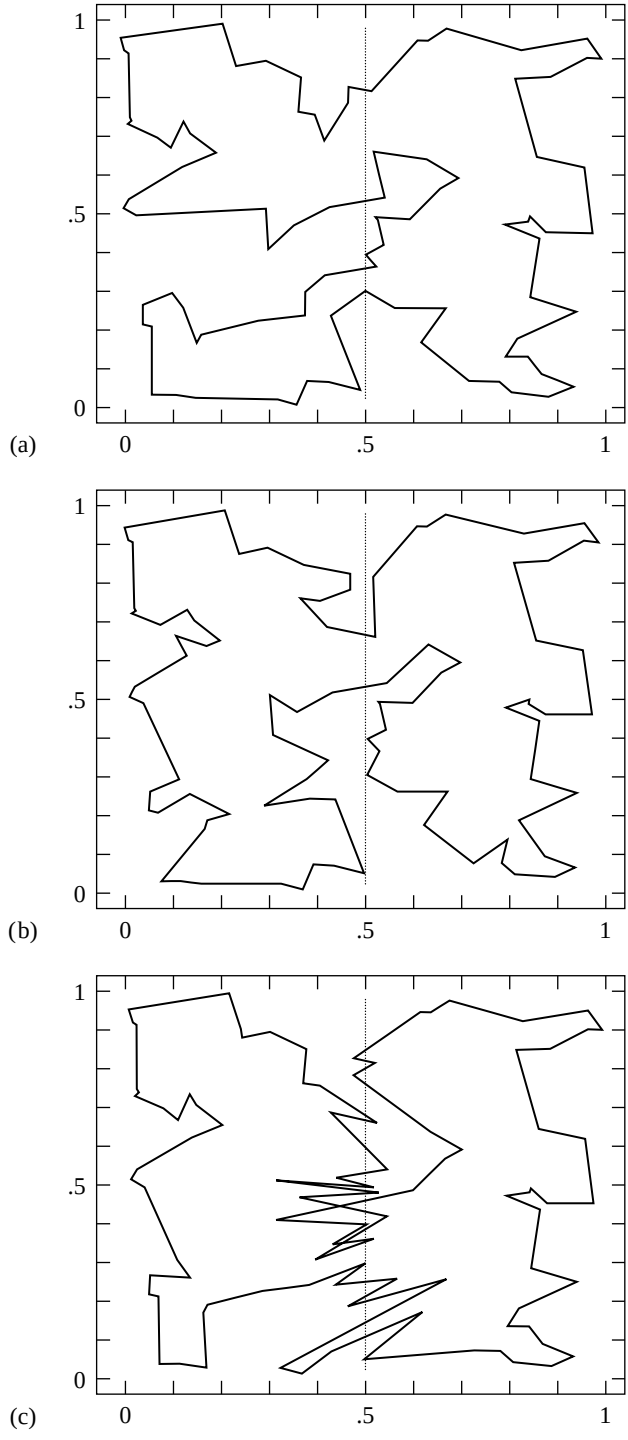


Figure 10.9.1. Traveling salesman problem solved by simulated annealing. The (nearly) shortest path among 100 randomly positioned cities is shown in (a). The dotted line is a river, but there is no penalty in crossing. In (b) the river-crossing penalty is made large, and the solution restricts itself to the minimum number of crossings, two. In (c) the penalty has been made negative: the salesman is actually a smuggler who crosses the river on the flimsiest excuse!

```
            nn=1+mod((n(1)-n(2)+ncity-1),ncity)      nn is the number of cities not on the
            if (nn.lt.3) goto 1                      segment.
            idec=irbit1(iseed)      Decide whether to do a segment reversal or transport.
            if (idec.eq.0) then     Do a transport.
                n(3)=n(2)+int(abs(nn-2)*ran3(idum))+1
                n(3)=1+mod(n(3)-1,ncity)          Transport to a location not on the path.
                call trncst(x,y,iorder,ncity,n,de)       Calculate cost.
                call metrop(de,t,ans)                    Consult the oracle.
                if (ans) then
                    nsucc=nsucc+1
                    path=path+de
                    call trnspt(iorder,ncity,n)          Carry out the transport.
                endif
            else                                         Do a path reversal.
                call revcst(x,y,iorder,ncity,n,de)       Calculate cost.
                call metrop(de,t,ans)                    Consult the oracle.
                if (ans) then
                    nsucc=nsucc+1
                    path=path+de
                    call revers(iorder,ncity,n)          Carry out the reversal.
                endif
            endif
            if (nsucc.ge.nlimit) goto 2              Finish early if we have enough
        enddo 12                                         successful changes.
2       write(*,*)
        write(*,*) 'T =',t,' Path Length =',path
        write(*,*) 'Successful Moves: ',nsucc
        t=t*tfactr                                   Annealing schedule.
        if (nsucc.eq.0) return                       If no success, we are done.
    enddo 13
    return
    END
```

```
    SUBROUTINE revcst(x,y,iorder,ncity,n,de)
    INTEGER ncity,iorder(ncity),n(6)
    REAL de,x(ncity),y(ncity)
        This subroutine returns the value of the cost function for a proposed path reversal. ncity
        is the number of cities, and arrays x(1:ncity),y(1:ncity) give the coordinates of these
        cities. iorder(1:ncity) holds the present itinerary. The first two values n(1) and n(2)
        of array n give the starting and ending cities along the path segment which is to be reversed.
        On output, de is the cost of making the reversal. The actual reversal is not performed by
        this routine.
    INTEGER ii,j
    REAL alen,xx(4),yy(4),x1,x2,y1,y2
    alen(x1,x2,y1,y2)=sqrt((x2-x1)**2+(y2-y1)**2)
    n(3)=1+mod((n(1)+ncity-2),ncity)          Find the city before n(1) ..
    n(4)=1+mod(n(2),ncity)                        .. and the city after n(2).
    do 11 j=1,4
        ii=iorder(n(j))                       Find coordinates for the four cities involved.
        xx(j)=x(ii)
        yy(j)=y(ii)
    enddo 11
    de=-alen(xx(1),xx(3),yy(1),yy(3))         Calculate cost of disconnecting the segment
*       -alen(xx(2),xx(4),yy(2),yy(4))            at both ends and reconnecting in the op-
*       +alen(xx(1),xx(4),yy(1),yy(4))            posite order.
*       +alen(xx(2),xx(3),yy(2),yy(3))
    return
    END
```

```
      SUBROUTINE revers(iorder,ncity,n)
      INTEGER ncity,iorder(ncity),n(6)
```

This routine performs a path segment reversal. iorder(1:ncity) is an input array giving the present itinerary. The vector n has as its first four elements the first and last cities n(1),n(2) of the path segment to be reversed, and the two cities n(3) and n(4) that immediately precede and follow this segment. n(3) and n(4) are found by subroutine revcst. On output, iorder(1:ncity) contains the segment from n(1) to n(2) in reversed order.

```
      INTEGER itmp,j,k,l,nn
      nn=(1+mod(n(2)-n(1)+ncity,ncity))/2        This many cities must be swapped to effect
      do 11 j=1,nn                                  the reversal.
        k=1+mod((n(1)+j-2),ncity)                Start at the ends of the segment and swap
        l=1+mod((n(2)-j+ncity),ncity)               pairs of cities, moving toward the cen-
        itmp=iorder(k)                              ter.
        iorder(k)=iorder(l)
        iorder(l)=itmp
      enddo 11
      return
      END
```

```
      SUBROUTINE trncst(x,y,iorder, ncity,n,de)
      INTEGER ncity,iorder(ncity),n(6)
      REAL de,x(ncity),y(ncity)
```

This subroutine returns the value of the cost function for a proposed path segment transport. ncity is the number of cities, and arrays x(1:ncity) and y(1:ncity) give the city coordinates. iorder is an array giving the present itinerary. The first three elements of array n give the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, de is the cost of the change. The actual transport is not performed by this routine.

```
      INTEGER ii,j
      REAL xx(6),yy(6),alen,x1,x2,y1,y2
      alen(x1,x2,y1,y2)=sqrt((x2-x1)**2+(y2-y1)**2)
      n(4)=1+mod(n(3),ncity)                     Find the city following n(3)..
      n(5)=1+mod((n(1)+ncity-2),ncity)           ..and the one preceding n(1)..
      n(6)=1+mod(n(2),ncity)                     ..and the one following n(2).
      do 11 j=1,6
        ii=iorder(n(j))                          Determine coordinates for the six cities in-
        xx(j)=x(ii)                                 volved.
        yy(j)=y(ii)
      enddo 11
      de=-alen(xx(2),xx(6),yy(2),yy(6))          Calculate the cost of disconnecting the path
     *    -alen(xx(1),xx(5),yy(1),yy(5))            segment from n(1) to n(2), opening a
     *    -alen(xx(3),xx(4),yy(3),yy(4))            space between n(3) and n(4), connect-
     *    +alen(xx(1),xx(3),yy(1),yy(3))            ing the segment in the space, and con-
     *    +alen(xx(2),xx(4),yy(2),yy(4))            necting n(5) to n(6).
     *    +alen(xx(5),xx(6),yy(5),yy(6))
      return
      END
```

```
      SUBROUTINE trnspt(iorder,ncity,n)
      INTEGER ncity,iorder(ncity),n(6),MXCITY
      PARAMETER (MXCITY=1000)                    Maximum number of cities anticipated.
```

This routine does the actual path transport, once metrop has approved. iorder is an input array of length ncity giving the present itinerary. The array n has as its six elements the beginning n(1) and end n(2) of the path to be transported, the adjacent cities n(3) and n(4) between which the path is to be placed, and the cities n(5) and n(6) that precede and follow the path. n(4), n(5), and n(6) are calculated by subroutine trncst. On output, iorder is modified to reflect the movement of the path segment.

```
      INTEGER j,jj,m1,m2,m3,nn,jorder(MXCITY)
      m1=1+mod((n(2)-n(1)+ncity),ncity)          Find number of cities from n(1) to n(2)
```

```
      m2=1+mod((n(5)-n(4)+ncity),ncity)          ...and the number from n(4) to n(5)
      m3=1+mod((n(3)-n(6)+ncity),ncity)          ...and the number from n(6) to n(3).
      nn=1
      do 11 j=1,m1
        jj=1+mod((j+n(1)-2),ncity)               Copy the chosen segment.
        jorder(nn)=iorder(jj)
        nn=nn+1
      enddo 11
      do 12 j=1,m2                               Then copy the segment from n(4) to n(5).
        jj=1+mod((j+n(4)-2),ncity)
        jorder(nn)=iorder(jj)
        nn=nn+1
      enddo 12
      do 13 j=1,m3                               Finally, the segment from n(6) to n(3).
        jj=1+mod((j+n(6)-2),ncity)
        jorder(nn)=iorder(jj)
        nn=nn+1
      enddo 13
      do 14 j=1,ncity
        iorder(j)=jorder(j)                      Copy jorder back into iorder.
      enddo 14
      return
      END
```

```
      SUBROUTINE metrop(de,t,ans)
      REAL de,t
      LOGICAL ans
C     USES ran3
        Metropolis algorithm. ans is a logical variable that issues a verdict on whether to accept a
        reconfiguration that leads to a change de in the objective function e. If de<0, ans=.true.,
        while if de>0, ans is only .true. with probability exp(-de/t), where t is a temperature
        determined by the annealing schedule.
      INTEGER jdum
      REAL ran3
      SAVE jdum
      DATA jdum /1/
      ans=(de.lt.0.0).or.(ran3(jdum).lt.exp(-de/t))
      return
      END
```

Continuous Minimization by Simulated Annealing

The basic ideas of simulated annealing are also applicable to optimization problems with continuous N-dimensional control spaces, e.g., finding the (ideally, global) minimum of some function $f(\mathbf{x})$, in the presence of many local minima, where $\mathbf{x}$ is an N-dimensional vector. The four elements required by the Metropolis procedure are now as follows: The value of f is the objective function. The system state is the point $\mathbf{x}$. The control parameter T is, as before, something like a temperature, with an annealing schedule by which it is gradually reduced. And there must be a generator of random changes in the configuration, that is, a procedure for taking a random step from $\mathbf{x}$ to $\mathbf{x} + \Delta\mathbf{x}$.

The last of these elements is the most problematical. The literature to date [7-10] describes several different schemes for choosing $\Delta\mathbf{x}$, none of which, in our view, inspire complete confidence. The problem is one of efficiency: A generator of random changes is inefficient if, *when local downhill moves exist*, it nevertheless almost always proposes an uphill move. A good generator, we think, should not become inefficient in narrow valleys; nor should it become more and more inefficient as convergence to a minimum is approached. Except possibly for [7], all of the schemes that we have seen are inefficient in one or both of these situations.

Our own way of doing simulated annealing minimization on continuous control spaces is to use a modification of the downhill simplex method (§10.4). This amounts to replacing the single point $\mathbf{x}$ as a description of the system state by a simplex of $N+1$ points. The "moves" are the same as described in §10.4, namely reflections, expansions, and contractions of the simplex. The implementation of the Metropolis procedure is slightly subtle: We *add* a positive, logarithmically distributed random variable, proportional to the temperature T, to the stored function value associated with every vertex of the simplex, and we *subtract* a similar random variable from the function value of every new point that is tried as a replacement point. Like the ordinary Metropolis procedure, this method always accepts a true downhill step, but sometimes accepts an uphill one. In the limit $T \to 0$, this algorithm reduces exactly to the downhill simplex method and converges to a local minimum.

At a finite value of T, the simplex expands to a scale that approximates the size of the region that can be reached at this temperature, and then executes a stochastic, tumbling Brownian motion within that region, sampling new, approximately random, points as it does so. The efficiency with which a region is explored is independent of its narrowness (for an ellipsoidal valley, the ratio of its principal axes) and orientation. If the temperature is reduced sufficiently slowly, it becomes highly likely that the simplex will shrink into that region containing the lowest relative minimum encountered.

As in all applications of simulated annealing, there can be quite a lot of problem-dependent subtlety in the phrase "sufficiently slowly"; success or failure is quite often determined by the choice of annealing schedule. Here are some possibilities worth trying:

- Reduce T to $(1-\epsilon)T$ after every m moves, where ϵ/m is determined by experiment.
- Budget a total of K moves, and reduce T after every m moves to a value $T = T_0(1-k/K)^\alpha$, where k is the cumulative number of moves thus far, and α is a constant, say 1, 2, or 4. The optimal value for α depends on the statistical distribution of relative minima of various depths. Larger values of α spend more iterations at lower temperature.
- After every m moves, set T to β times $f_1 - f_b$, where β is an experimentally determined constant of order 1, f_1 is the smallest function value currently represented in the simplex, and f_b is the best function ever encountered. However, never reduce T by more than some fraction γ at a time.

Another strategic question is whether to do an occasional *restart*, where a vertex of the simplex is discarded in favor of the "best-ever" point. (You must be sure that the best-ever point is not currently in the simplex when you do this!) We have found problems for which restarts — every time the temperature has decreased by a factor of 3, say — are highly beneficial; we have found other problems for which restarts

have no positive, or a somewhat negative, effect.

You should compare the following routine, amebsa, with its counterpart amoeba in §10.4. Note that the argument iter is used in a somewhat different manner.

```
      SUBROUTINE amebsa(p,y,mp,np,ndim,pb,yb,ftol,funk,iter,temptr)
      INTEGER iter,mp,ndim,np,NMAX
      REAL ftol,temptr,yb,p(mp,np),pb(np),y(mp),funk
      PARAMETER (NMAX=200)
      EXTERNAL funk
C     USES amotsa,funk,ran1
        Multidimensional minimization of the function funk(x) where x(1:ndim) is a vector in
        ndim dimensions, by simulated annealing combined with the downhill simplex method of
        Nelder and Mead. The input matrix p(1..ndim+1,1..ndim) has ndim+1 rows, each an
        ndim-dimensional vector which is a vertex of the starting simplex. Also input is the vector
        y(1:ndim+1), whose components must be pre-initialized to the values of funk evaluated at
        the ndim+1 vertices (rows) of p; ftol, the fractional convergence tolerance to be achieved
        in the function value for an early return; iter, and temptr. The routine makes iter
        function evaluations at an annealing temperature temptr, then returns. You should then
        decrease temptr according to your annealing schedule, reset iter, and call the routine
        again (leaving other arguments unaltered between calls). If iter is returned with a positive
        value, then early convergence and return occurred. If you initialize yb to a very large value
        on the first call, then yb and pb(1:ndim) will subsequently return the best function value
        and point ever encountered (even if it is no longer a point in the simplex).
      INTEGER i,idum,ihi,ilo,j,m,n
      REAL rtol,sum,swap,tt,yhi,ylo,ynhi,ysave,yt,ytry,psum(NMAX),
     *     amotsa,ran1
      COMMON /ambsa/ tt,idum
      tt=-temptr
1     do 12 n=1,ndim                    Enter here when starting or after overall contraction.
        sum=0.                          Recompute psum.
        do 11 m=1,ndim+1
          sum=sum+p(m,n)
        enddo 11
        psum(n)=sum
      enddo 12
2     ilo=1                             Enter here after changing a single point. Find which point
      ihi=2                               is the highest (worst), next-highest, and lowest (best).
      ylo=y(1)+tt*log(ran1(idum))       Whenever we "look at" a vertex, it gets a random thermal
      ynhi=ylo                            fluctuation.
      yhi=y(2)+tt*log(ran1(idum))
      if (ylo.gt.yhi) then
        ihi=1
        ilo=2
        ynhi=yhi
        yhi=ylo
        ylo=ynhi
      endif
      do 13 i=3,ndim+1                  Loop over the points in the simplex.
        yt=y(i)+tt*log(ran1(idum))      More thermal fluctuations.
        if(yt.le.ylo) then
          ilo=i
          ylo=yt
        endif
        if(yt.gt.yhi) then
          ynhi=yhi
          ihi=i
          yhi=yt
        else if(yt.gt.ynhi) then
          ynhi=yt
        endif
      enddo 13
      rtol=2.*abs(yhi-ylo)/(abs(yhi)+abs(ylo))
```

```
      Compute the fractional range from highest to lowest and return if satisfactory.
    if (rtol.lt.ftol.or.iter.lt.0) then    If returning, put best point and value in slot 1.
        swap=y(1)
        y(1)=y(ilo)
        y(ilo)=swap
        do 14 n=1,ndim
            swap=p(1,n)
            p(1,n)=p(ilo,n)
            p(ilo,n)=swap
        enddo 14
        return
    endif
    iter=iter-2
      Begin a new iteration. First extrapolate by a factor −1 through the face of the simplex across
      from the high point, i.e., reflect the simplex from the high point.
    ytry=amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihi,yhi,-1.0)
    if (ytry.le.ylo) then
      Gives a result better than the best point, so try an additional extrapolation by a factor 2.
        ytry=amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihi,yhi,2.0)
    else if (ytry.ge.ynhi) then
      The reflected point is worse than the second-highest, so look for an intermediate lower point,
      i.e., do a one-dimensional contraction.
        ysave=yhi
        ytry=amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihi,yhi,0.5)
        if (ytry.ge.ysave) then      Can't seem to get rid of that high point. Better contract
            do 16 i=1,ndim+1             around the lowest (best) point.
                if(i.ne.ilo)then
                    do 15 j=1,ndim
                        psum(j)=0.5*(p(i,j)+p(ilo,j))
                        p(i,j)=psum(j)
                    enddo 15
                    y(i)=funk(psum)
                endif
            enddo 16
            iter=iter-ndim
            goto 1
        endif
    else
        iter=iter+1                  Correct the evaluation count.
    endif
    goto 2
    END
```

```
    FUNCTION amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihi,yhi,fac)
    INTEGER ihi,mp,ndim,np,NMAX
    REAL amotsa,fac,yb,yhi,p(mp,np),pb(np),psum(np),y(mp),funk
    PARAMETER (NMAX=200)
    EXTERNAL funk
C   USES funk,ran1
      Extrapolates by a factor fac through the face of the simplex across from the high point,
      tries it, and replaces the high point if the new point is better.
    INTEGER idum,j
    REAL fac1,fac2,tt,yflu,ytry,ptry(NMAX),ran1
    COMMON /ambsa/ tt,idum
    fac1=(1.-fac)/ndim
    fac2=fac1-fac
    do 11 j=1,ndim
        ptry(j)=psum(j)*fac1-p(ihi,j)*fac2
    enddo 11
```

```
ytry=funk(ptry)
if (ytry.le.yb) then                Save the best-ever.
    do 12 j=1,ndim
        pb(j)=ptry(j)
    enddo 12
    yb=ytry
endif
yflu=ytry-tt*log(ran1(idum))        We added a thermal fluctuation to all the current vertices,
if (yflu.lt.yhi) then                   but we subtract it here, so as to give the simplex
    y(ihi)=ytry                         a thermal Brownian motion: It likes to accept any
    yhi=yflu                            suggested change.
    do 13 j=1,ndim
        psum(j)=psum(j)-p(ihi,j)+ptry(j)
        p(ihi,j)=ptry(j)
    enddo 13
endif
amotsa=yflu
return
END
```

There is not yet enough practical experience with the method of simulated annealing to say definitively what its future place among optimization methods will be. The method has several extremely attractive features, rather unique when compared with other optimization techniques.

First, it is not "greedy," in the sense that it is not easily fooled by the quick payoff achieved by falling into unfavorable local minima. Provided that sufficiently general reconfigurations are given, it wanders freely among local minima of depth less than about T. As T is lowered, the number of such minima qualifying for frequent visits is gradually reduced.

Second, configuration decisions tend to proceed in a logical order. Changes that cause the greatest energy differences are sifted over when the control parameter T is large. These decisions become more permanent as T is lowered, and attention then shifts more to smaller refinements in the solution. For example, in the traveling salesman problem with the Mississippi River twist, if λ is large, a decision to cross the Mississippi only twice is made at high T, while the specific routes on each side of the river are determined only at later stages.

The analogies to thermodynamics may be pursued to a greater extent than we have done here. Quantities analogous to specific heat and entropy may be defined, and these can be useful in monitoring the progress of the algorithm towards an acceptable solution. Information on this subject is found in [1].

CITED REFERENCES AND FURTHER READING:

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. 1983, *Science*, vol. 220, pp. 671–680. [1]

Kirkpatrick, S. 1984, *Journal of Statistical Physics*, vol. 34, pp. 975–986. [2]

Vecchi, M.P. and Kirkpatrick, S. 1983, *IEEE Transactions on Computer Aided Design*, vol. CAD-2, pp. 215–222. [3]

Otten, R.H.J.M., and van Ginneken, L.P.P.P. 1989, *The Annealing Algorithm* (Boston: Kluwer) [contains many references to the literature]. [4]

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller A., and Teller, E. 1953, *Journal of Chemical Physics*, vol. 21, pp. 1087–1092. [5]

Lin, S. 1965, *Bell System Technical Journal*, vol. 44, pp. 2245–2269. [6]

Vanderbilt, D., and Louie, S.G. 1984, *Journal of Computational Physics*, vol. 56, pp. 259–271. [7]

Bohachevsky, I.O., Johnson, M.E., and Stein, M.L. 1986, *Technometrics*, vol. 28, pp. 209–217. [8]

Corana, A., Marchesi, M., Martini, C., and Ridella, S. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 262–280. [9]

Bélisle, C.J.P., Romeijn, H.E., and Smith, R.L. 1990, Technical Report 90–25, Department of Industrial and Operations Engineering, University of Michigan, submitted to *Mathematical Programming*. [10]

Christofides, N., Mingozzi, A., Toth, P., and Sandi, C. (eds.) 1979, *Combinatorial Optimization* (London and New York: Wiley-Interscience) [not simulated annealing, but other topics and algorithms].

Chapter 11. Eigensystems

11.0 Introduction

An $N \times N$ matrix $\mathbf{A}$ is said to have an *eigenvector* $\mathbf{x}$ and corresponding *eigenvalue* λ if

$$\mathbf{A} \cdot \mathbf{x} = \lambda \mathbf{x} \tag{11.0.1}$$

Obviously any multiple of an eigenvector $\mathbf{x}$ will also be an eigenvector, but we won't consider such multiples as being distinct eigenvectors. (The zero vector is not considered to be an eigenvector at all.) Evidently (11.0.1) can hold only if

$$\det |\mathbf{A} - \lambda \mathbf{1}| = 0 \tag{11.0.2}$$

which, if expanded out, is an Nth degree polynomial in λ whose roots are the eigenvalues. This proves that there are always N (not necessarily distinct) eigenvalues. Equal eigenvalues coming from multiple roots are called *degenerate*. Root-searching in the characteristic equation (11.0.2) is usually a very poor computational method for finding eigenvalues. We will learn much better ways in this chapter, as well as efficient ways for finding corresponding eigenvectors.

The above two equations also prove that every one of the N eigenvalues has a (not necessarily distinct) corresponding eigenvector: If λ is set to an eigenvalue, then the matrix $\mathbf{A} - \lambda \mathbf{1}$ is singular, and we know that every singular matrix has at least one nonzero vector in its nullspace (see §2.6 on singular value decomposition).

If you add $\tau \mathbf{x}$ to both sides of (11.0.1), you will easily see that the eigenvalues of any matrix can be changed or *shifted* by an additive constant τ by adding to the matrix that constant times the identity matrix. The eigenvectors are unchanged by this shift. Shifting, as we will see, is an important part of many algorithms for computing eigenvalues. We see also that there is no special significance to a zero eigenvalue. Any eigenvalue can be shifted to zero, or any zero eigenvalue can be shifted away from zero.

Definitions and Basic Facts

A matrix is called *symmetric* if it is equal to its transpose,

$$\mathbf{A} = \mathbf{A}^T \qquad \text{or} \qquad a_{ij} = a_{ji} \tag{11.0.3}$$

It is called *Hermitian* or *self-adjoint* if it equals the complex-conjugate of its transpose (its *Hermitian conjugate*, denoted by "†")

$$\mathbf{A} = \mathbf{A}^\dagger \qquad \text{or} \qquad a_{ij} = a_{ji}{}^* \tag{11.0.4}$$

It is termed *orthogonal* if its transpose equals its inverse,

$$\mathbf{A}^T \cdot \mathbf{A} = \mathbf{A} \cdot \mathbf{A}^T = \mathbf{1} \tag{11.0.5}$$

and *unitary* if its Hermitian conjugate equals its inverse. Finally, a matrix is called *normal* if it *commutes* with its Hermitian conjugate,

$$\mathbf{A} \cdot \mathbf{A}^\dagger = \mathbf{A}^\dagger \cdot \mathbf{A} \tag{11.0.6}$$

For real matrices, Hermitian means the same as symmetric, unitary means the same as orthogonal, and *both* of these distinct classes are normal.

The reason that "Hermitian" is an important concept has to do with eigenvalues. The eigenvalues of a Hermitian matrix are all real. In particular, the eigenvalues of a real symmetric matrix are all real. Contrariwise, the eigenvalues of a real nonsymmetric matrix may include real values, but may also include pairs of complex conjugate values; and the eigenvalues of a complex matrix that is not Hermitian will in general be complex.

The reason that "normal" is an important concept has to do with the eigenvectors. The eigenvectors of a normal matrix with nondegenerate (i.e., distinct) eigenvalues are complete and orthogonal, spanning the N-dimensional vector space. For a normal matrix with degenerate eigenvalues, we have the additional freedom of replacing the eigenvectors corresponding to a degenerate eigenvalue by linear combinations of themselves. Using this freedom, we can always perform Gram-Schmidt orthogonalization (consult any linear algebra text) and *find* a set of eigenvectors that are complete and orthogonal, just as in the nondegenerate case. The matrix whose columns are an orthonormal set of eigenvectors is evidently unitary. A special case is that the matrix of eigenvectors of a real, symmetric matrix is orthogonal, since the eigenvectors of that matrix are all real.

When a matrix is not normal, as typified by any random, nonsymmetric, real matrix, then in general we cannot find *any* orthonormal set of eigenvectors, nor even any pairs of eigenvectors that are orthogonal (except perhaps by rare chance). While the N non-orthonormal eigenvectors will "usually" span the N-dimensional vector space, they do not always do so; that is, the eigenvectors are not always complete. Such a matrix is said to be *defective*.

Left and Right Eigenvectors

While the eigenvectors of a non-normal matrix are not particularly orthogonal among themselves, they *do* have an orthogonality relation with a different set of vectors, which we must now define. Up to now our eigenvectors have been column vectors that are multiplied to the right of a matrix $\mathbf{A}$, as in (11.0.1). These, more explicitly, are termed *right eigenvectors*. We could also, however, try to find row vectors, which multiply $\mathbf{A}$ to the left and satisfy

$$\mathbf{x} \cdot \mathbf{A} = \lambda \mathbf{x} \tag{11.0.7}$$

These are called *left eigenvectors*. By taking the transpose of equation (11.0.7), we see that every left eigenvector is the transpose of a right eigenvector *of the transpose of* $\mathbf{A}$. Now by comparing to (11.0.2), and using the fact that the determinant of a matrix equals the determinant of its transpose, we also see that the left and right eigen*values* of $\mathbf{A}$ are identical.

If the matrix $\mathbf{A}$ is symmetric, then the left and right eigenvectors are just transposes of each other, that is, have the same numerical values as components. Likewise, if the matrix is self-adjoint, the left and right eigenvectors are Hermitian conjugates of each other. For the general nonnormal case, however, we have the following calculation: Let $\mathbf{X}_R$ be the matrix formed by columns from the right eigenvectors, and $\mathbf{X}_L$ be the matrix formed by rows from the left eigenvectors. Then (11.0.1) and (11.0.7) can be rewritten as

$$\mathbf{A} \cdot \mathbf{X}_R = \mathbf{X}_R \cdot \mathrm{diag}(\lambda_1 \dots \lambda_N) \qquad \mathbf{X}_L \cdot \mathbf{A} = \mathrm{diag}(\lambda_1 \dots \lambda_N) \cdot \mathbf{X}_L \tag{11.0.8}$$

Multiplying the first of these equations on the left by $\mathbf{X}_L$, the second on the right by $\mathbf{X}_R$, and subtracting the two, gives

$$(\mathbf{X}_L \cdot \mathbf{X}_R) \cdot \mathrm{diag}(\lambda_1 \dots \lambda_N) = \mathrm{diag}(\lambda_1 \dots \lambda_N) \cdot (\mathbf{X}_L \cdot \mathbf{X}_R) \tag{11.0.9}$$

This says that the matrix of dot products of the left and right eigenvectors commutes with the diagonal matrix of eigenvalues. But the only matrices that commute with a diagonal matrix *of distinct elements* are themselves diagonal. Thus, if the eigenvalues are nondegenerate, each left eigenvector is orthogonal to all right eigenvectors except its corresponding one, and vice versa. By choice of normalization, the dot products of corresponding left and right eigenvectors can always be made unity for any matrix with nondegenerate eigenvalues.

If some eigenvalues are degenerate, then either the left or the right eigenvectors corresponding to a degenerate eigenvalue must be linearly combined among themselves to achieve orthogonality with the right or left ones, respectively. This can always be done by a procedure akin to Gram-Schmidt orthogonalization. The normalization can then be adjusted to give unity for the nonzero dot products between corresponding left and right eigenvectors. If the dot product of corresponding left and right eigenvectors is zero at this stage, then you have a case where the eigenvectors are incomplete! Note that incomplete eigenvectors can occur only where there are degenerate eigenvalues, but do not always occur in such cases (in fact, never occur for the class of "normal" matrices). See [1] for a clear discussion.

In both the degenerate and nondegenerate cases, the final normalization to unity of all nonzero dot products produces the result: The matrix whose rows are left eigenvectors is the inverse matrix of the matrix whose columns are right eigenvectors, *if the inverse exists.*

Diagonalization of a Matrix

Multiplying the first equation in (11.0.8) by $\mathbf{X}_L$, and using the fact that $\mathbf{X}_L$ and $\mathbf{X}_R$ are matrix inverses, we get

$$\mathbf{X}_R^{-1} \cdot \mathbf{A} \cdot \mathbf{X}_R = \text{diag}(\lambda_1 \dots \lambda_N) \tag{11.0.10}$$

This is a particular case of a *similarity transform* of the matrix $\mathbf{A}$,

$$\mathbf{A} \quad \rightarrow \quad \mathbf{Z}^{-1} \cdot \mathbf{A} \cdot \mathbf{Z} \tag{11.0.11}$$

for some transformation matrix $\mathbf{Z}$. Similarity transformations play a crucial role in the computation of eigenvalues, because they leave the eigenvalues of a matrix unchanged. This is easily seen from

$$\begin{aligned} \det\left|\mathbf{Z}^{-1} \cdot \mathbf{A} \cdot \mathbf{Z} - \lambda\mathbf{1}\right| &= \det\left|\mathbf{Z}^{-1} \cdot (\mathbf{A} - \lambda\mathbf{1}) \cdot \mathbf{Z}\right| \\ &= \det|\mathbf{Z}|\ \det|\mathbf{A} - \lambda\mathbf{1}|\ \det\left|\mathbf{Z}^{-1}\right| \\ &= \det|\mathbf{A} - \lambda\mathbf{1}| \end{aligned} \tag{11.0.12}$$

Equation (11.0.10) shows that any matrix with complete eigenvectors (which includes all normal matrices and "most" random nonnormal ones) can be diagonalized by a similarity transformation, that the columns of the transformation matrix that effects the diagonalization are the right eigenvectors, and that the rows of its inverse are the left eigenvectors.

For real, symmetric matrices, the eigenvectors are real and orthonormal, so the transformation matrix is orthogonal. The similarity transformation is then also an *orthogonal transformation* of the form

$$\mathbf{A} \quad \rightarrow \quad \mathbf{Z}^T \cdot \mathbf{A} \cdot \mathbf{Z} \tag{11.0.13}$$

While real nonsymmetric matrices can be diagonalized in their usual case of complete eigenvectors, the transformation matrix is not necessarily real. It turns out, however, that a real similarity transformation can "almost" do the job. It can reduce the matrix down to a form with little two-by-two blocks along the diagonal, all other elements zero. Each two-by-two block corresponds to a complex-conjugate pair of complex eigenvalues. We will see this idea exploited in some routines given later in the chapter.

The "grand strategy" of virtually all modern eigensystem routines is to nudge the matrix $\mathbf{A}$ towards diagonal form by a sequence of similarity transformations,

$$\begin{aligned} \mathbf{A} \quad &\rightarrow \quad \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \quad \rightarrow \quad \mathbf{P}_2^{-1} \cdot \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \cdot \mathbf{P}_2 \\ &\rightarrow \quad \mathbf{P}_3^{-1} \cdot \mathbf{P}_2^{-1} \cdot \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \quad \rightarrow \quad \text{etc.} \end{aligned} \tag{11.0.14}$$

If we get all the way to diagonal form, then the eigenvectors are the columns of the accumulated transformation

$$\mathbf{X}_R = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdot \dots \tag{11.0.15}$$

Sometimes we do not want to go all the way to diagonal form. For example, if we are interested only in eigenvalues, not eigenvectors, it is enough to transform the matrix $\mathbf{A}$ to be triangular, with all elements below (or above) the diagonal zero. In this case the diagonal elements are already the eigenvalues, as you can see by mentally evaluating (11.0.2) using expansion by minors.

There are two rather different sets of techniques for implementing the grand strategy (11.0.14). It turns out that they work rather well in combination, so most modern eigensystem routines use both. The first set of techniques constructs individual $\mathbf{P}_i$'s as explicit "atomic" transformations designed to perform specific tasks, for example zeroing a particular off-diagonal element (Jacobi transformation, §11.1), or a whole particular row or column (Householder transformation, §11.2; elimination method, §11.5). In general, a finite sequence of these simple transformations cannot completely diagonalize a matrix. There are then two choices: either use the finite sequence of transformations to go most of the way (e.g., to some special form like *tridiagonal* or *Hessenberg*, see §11.2 and §11.5 below) and follow up with the second set of techniques about to be mentioned; or else iterate the finite sequence of simple transformations over and over until the deviation of the matrix from diagonal is negligibly small. This latter approach is conceptually simplest, so we will discuss it in the next section; however, for N greater than ~ 10, it is computationally inefficient by a roughly constant factor ~ 5.

The second set of techniques, called *factorization methods*, is more subtle. Suppose that the matrix $\mathbf{A}$ can be factored into a left factor $\mathbf{F}_L$ and a right factor $\mathbf{F}_R$. Then

$$\mathbf{A} = \mathbf{F}_L \cdot \mathbf{F}_R \qquad \text{or equivalently} \qquad \mathbf{F}_L^{-1} \cdot \mathbf{A} = \mathbf{F}_R \tag{11.0.16}$$

If we now multiply back together the factors in the reverse order, and use the second equation in (11.0.16) we get

$$\mathbf{F}_R \cdot \mathbf{F}_L = \mathbf{F}_L^{-1} \cdot \mathbf{A} \cdot \mathbf{F}_L \tag{11.0.17}$$

which we recognize as having effected a similarity transformation on $\mathbf{A}$ with the transformation matrix being $\mathbf{F}_L$! In §11.3 and §11.6 we will discuss the *QR method* which exploits this idea.

Factorization methods also do not converge exactly in a finite number of transformations. But the better ones do converge rapidly and reliably, and, when following an appropriate initial reduction by simple similarity transformations, they are the methods of choice.

"Eigenpackages of Canned Eigenroutines"

You have probably gathered by now that the solution of eigensystems is a fairly complicated business. It is. It is one of the few subjects covered in this book for which we do *not* recommend that you avoid canned routines. On the contrary, the purpose of this chapter is precisely to give you some appreciation of what is going on inside such canned routines, so that you can make intelligent choices about using them, and intelligent diagnoses when something goes wrong.

You will find that almost all canned routines in use nowadays trace their ancestry back to routines published in Wilkinson and Reinsch's *Handbook for Automatic Computation, Vol. II, Linear Algebra* [2]. This excellent reference, containing papers by a number of authors, is the Bible of the field. A public-domain implementation of the *Handbook* routines in FORTRAN is the EISPACK set of programs [3]. The routines in this chapter are translations of either the *Handbook* or EISPACK routines, so understanding these will take you a lot of the way towards understanding those canonical packages.

IMSL [4] and NAG [5] each provide proprietary implementations, in FORTRAN, of what are essentially the Handbook routines.

A good "eigenpackage" will provide separate routines, or separate paths through sequences of routines, for the following desired calculations:

- all eigenvalues and no eigenvectors
- all eigenvalues and some corresponding eigenvectors
- all eigenvalues and all corresponding eigenvectors

The purpose of these distinctions is to save compute time and storage; it is wasteful to calculate eigenvectors that you don't need. Often one is interested only in the eigenvectors corresponding to the largest few eigenvalues, or largest few in magnitude, or few that are negative. The method usually used to calculate "some" eigenvectors is typically more efficient than calculating all eigenvectors if you desire fewer than about a quarter of the eigenvectors.

A good eigenpackage also provides separate paths for each of the above calculations for each of the following special forms of the matrix:

- real, symmetric, tridiagonal
- real, symmetric, banded (only a small number of sub- and superdiagonals are nonzero)
- real, symmetric
- real, nonsymmetric
- complex, Hermitian
- complex, non-Hermitian

Again, the purpose of these distinctions is to save time and storage by using the *least* general routine that will serve in any particular application.

In this chapter, as a bare introduction, we give good routines for the following paths:

- all eigenvalues and eigenvectors of a real, symmetric, tridiagonal matrix (§11.3)
- all eigenvalues and eigenvectors of a real, symmetric, matrix (§11.1–§11.3)
- all eigenvalues and eigenvectors of a complex, Hermitian matrix (§11.4)
- all eigenvalues and no eigenvectors of a real, nonsymmetric matrix

(§11.5–§11.6)

We also discuss, in §11.7, how to obtain some eigenvectors of nonsymmetric matrices by the method of inverse iteration.

Generalized and Nonlinear Eigenvalue Problems

Many eigenpackages also deal with the so-called *generalized eigenproblem*, [6]

$$\mathbf{A} \cdot \mathbf{x} = \lambda \mathbf{B} \cdot \mathbf{x} \tag{11.0.18}$$

where $\mathbf{A}$ and $\mathbf{B}$ are both matrices. Most such problems, where $\mathbf{B}$ is nonsingular, can be handled by the equivalent

$$(\mathbf{B}^{-1} \cdot \mathbf{A}) \cdot \mathbf{x} = \lambda \mathbf{x} \tag{11.0.19}$$

Often $\mathbf{A}$ and $\mathbf{B}$ are symmetric and $\mathbf{B}$ is positive definite. The matrix $\mathbf{B}^{-1} \cdot \mathbf{A}$ in (11.0.19) is not symmetric, but we can recover a symmetric eigenvalue problem by using the Cholesky decomposition $\mathbf{B} = \mathbf{L} \cdot \mathbf{L}^T$ of §2.9. Multiplying equation (11.0.18) by $\mathbf{L}^{-1}$, we get

$$\mathbf{C} \cdot (\mathbf{L}^T \cdot \mathbf{x}) = \lambda (\mathbf{L}^T \cdot \mathbf{x}) \tag{11.0.20}$$

where

$$\mathbf{C} = \mathbf{L}^{-1} \cdot \mathbf{A} \cdot (\mathbf{L}^{-1})^T \tag{11.0.21}$$

The matrix $\mathbf{C}$ is symmetric and its eigenvalues are the same as those of the original problem (11.0.18); its eigenfunctions are $\mathbf{L}^T \cdot \mathbf{x}$. The efficient way to form $\mathbf{C}$ is first to solve the equation

$$\mathbf{Y} \cdot \mathbf{L}^T = \mathbf{A} \tag{11.0.22}$$

for the lower triangle of the matrix $\mathbf{Y}$. Then solve

$$\mathbf{L} \cdot \mathbf{C} = \mathbf{Y} \tag{11.0.23}$$

for the lower triangle of the symmetric matrix $\mathbf{C}$.

Another generalization of the standard eigenvalue problem is to problems nonlinear in the eigenvalue λ, for example,

$$(\mathbf{A}\lambda^2 + \mathbf{B}\lambda + \mathbf{C}) \cdot \mathbf{x} = 0 \tag{11.0.24}$$

This can be turned into a linear problem by introducing an additional unknown eigenvector $\mathbf{y}$ and solving the $2N \times 2N$ eigensystem,

$$\begin{pmatrix} 0 & \mathbf{1} \\ -\mathbf{A}^{-1} \cdot \mathbf{C} & -\mathbf{A}^{-1} \cdot \mathbf{B} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \tag{11.0.25}$$

This technique generalizes to higher-order polynomials in λ. A polynomial of degree M produces a linear $MN \times MN$ eigensystem (see [7]).

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 6. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [4]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.), Chapter F02. [5]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §7.7. [6]

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [7]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 13.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

11.1 Jacobi Transformations of a Symmetric Matrix

The Jacobi method consists of a sequence of orthogonal similarity transformations of the form of equation (11.0.14). Each transformation (a *Jacobi rotation*) is just a plane rotation designed to annihilate one of the off-diagonal matrix elements. Successive transformations undo previously set zeros, but the off-diagonal elements nevertheless get smaller and smaller, until the matrix is diagonal to machine precision. Accumulating the product of the transformations as you go gives the matrix of eigenvectors, equation (11.0.15), while the elements of the final diagonal matrix are the eigenvalues.

The Jacobi method is absolutely foolproof for all real symmetric matrices. For matrices of order greater than about 10, say, the algorithm is slower, by a significant constant factor, than the QR method we shall give in §11.3. However, the Jacobi algorithm is much simpler than the more efficient methods. We thus recommend it for matrices of moderate order, where expense is not a major consideration.

The basic Jacobi rotation $\mathbf{P}_{pq}$ is a matrix of the form

$$\mathbf{P}_{pq} = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{bmatrix} \tag{11.1.1}$$

Here all the diagonal elements are unity except for the two elements c in rows (and columns) p and q. All off-diagonal elements are zero except the two elements s and $-s$. The numbers c and s are the cosine and sine of a rotation angle ϕ, so $c^2 + s^2 = 1$.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 6. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [4]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.), Chapter F02. [5]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §7.7. [6]

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [7]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 13.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

11.1 Jacobi Transformations of a Symmetric Matrix

The Jacobi method consists of a sequence of orthogonal similarity transformations of the form of equation (11.0.14). Each transformation (a *Jacobi rotation*) is just a plane rotation designed to annihilate one of the off-diagonal matrix elements. Successive transformations undo previously set zeros, but the off-diagonal elements nevertheless get smaller and smaller, until the matrix is diagonal to machine precision. Accumulating the product of the transformations as you go gives the matrix of eigenvectors, equation (11.0.15), while the elements of the final diagonal matrix are the eigenvalues.

The Jacobi method is absolutely foolproof for all real symmetric matrices. For matrices of order greater than about 10, say, the algorithm is slower, by a significant constant factor, than the QR method we shall give in §11.3. However, the Jacobi algorithm is much simpler than the more efficient methods. We thus recommend it for matrices of moderate order, where expense is not a major consideration.

The basic Jacobi rotation $\mathbf{P}_{pq}$ is a matrix of the form

$$\mathbf{P}_{pq} = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{bmatrix} \tag{11.1.1}$$

Here all the diagonal elements are unity except for the two elements c in rows (and columns) p and q. All off-diagonal elements are zero except the two elements s and $-s$. The numbers c and s are the cosine and sine of a rotation angle ϕ, so $c^2 + s^2 = 1$.

A plane rotation such as (11.1.1) is used to transform the matrix $\mathbf{A}$ according to

$$\mathbf{A}' = \mathbf{P}_{pq}^T \cdot \mathbf{A} \cdot \mathbf{P}_{pq} \tag{11.1.2}$$

Now, $\mathbf{P}_{pq}^T \cdot \mathbf{A}$ changes only rows p and q of $\mathbf{A}$, while $\mathbf{A} \cdot \mathbf{P}_{pq}$ changes only columns p and q. Notice that the subscripts p and q do not denote components of $\mathbf{P}_{pq}$, but rather label which kind of rotation the matrix is, i.e., which rows and columns it affects. Thus the changed elements of $\mathbf{A}$ in (11.1.2) are only in the p and q rows and columns indicated below:

$$\mathbf{A}' = \begin{bmatrix} & \cdots & a'_{1p} & \cdots & a'_{1q} & \cdots & \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{p1} & \cdots & a'_{pp} & \cdots & a'_{pq} & \cdots & a'_{pn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{q1} & \cdots & a'_{qp} & \cdots & a'_{qq} & \cdots & a'_{qn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ & \cdots & a'_{np} & \cdots & a'_{nq} & \cdots & \end{bmatrix} \tag{11.1.3}$$

Multiplying out equation (11.1.2) and using the symmetry of $\mathbf{A}$, we get the explicit formulas

$$\begin{aligned} a'_{rp} &= ca_{rp} - sa_{rq} \\ a'_{rq} &= ca_{rq} + sa_{rp} \end{aligned} \qquad r \neq p,\ r \neq q \tag{11.1.4}$$

$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sca_{pq} \tag{11.1.5}$$

$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq} \tag{11.1.6}$$

$$a'_{pq} = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \tag{11.1.7}$$

The idea of the Jacobi method is to try to zero the off-diagonal elements by a series of plane rotations. Accordingly, to set $a'_{pq} = 0$, equation (11.1.7) gives the following expression for the rotation angle ϕ

$$\theta \equiv \cot 2\phi \equiv \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}} \tag{11.1.8}$$

If we let $t \equiv s/c$, the definition of θ can be rewritten

$$t^2 + 2t\theta - 1 = 0 \tag{11.1.9}$$

The smaller root of this equation corresponds to a rotation angle less than $\pi/4$ in magnitude; this choice at each stage gives the most stable reduction. Using the form of the quadratic formula with the discriminant in the denominator, we can write this smaller root as

$$t = \frac{\text{sgn}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \tag{11.1.10}$$

If θ is so large that θ^2 would overflow on the computer, we set $t = 1/(2\theta)$. It now follows that

$$c = \frac{1}{\sqrt{t^2+1}} \tag{11.1.11}$$

$$s = tc \tag{11.1.12}$$

When we actually use equations (11.1.4)–(11.1.7) numerically, we rewrite them to minimize roundoff error. Equation (11.1.7) is replaced by

$$a'_{pq} = 0 \tag{11.1.13}$$

The idea in the remaining equations is to set the new quantity equal to the old quantity plus a small correction. Thus we can use (11.1.7) and (11.1.13) to eliminate a_{qq} from (11.1.5), giving

$$a'_{pp} = a_{pp} - ta_{pq} \tag{11.1.14}$$

Similarly,

$$a'_{qq} = a_{qq} + ta_{pq} \tag{11.1.15}$$

$$a'_{rp} = a_{rp} - s(a_{rq} + \tau a_{rp}) \tag{11.1.16}$$

$$a'_{rq} = a_{rq} + s(a_{rp} - \tau a_{rq}) \tag{11.1.17}$$

where τ $(= \tan\phi/2)$ is defined by

$$\tau \equiv \frac{s}{1+c} \tag{11.1.18}$$

One can see the convergence of the Jacobi method by considering the sum of the squares of the off-diagonal elements

$$S = \sum_{r \neq s} |a_{rs}|^2 \tag{11.1.19}$$

Equations (11.1.4)–(11.1.7) imply that

$$S' = S - 2|a_{pq}|^2 \tag{11.1.20}$$

(Since the transformation is orthogonal, the sum of the squares of the diagonal elements increases correspondingly by $2|a_{pq}|^2$.) The sequence of S's thus decreases monotonically. Since the sequence is bounded below by zero, and since we can choose a_{pq} to be whatever element we want, the sequence can be made to converge to zero.

Eventually one obtains a matrix $\mathbf{D}$ that is diagonal to machine precision. The diagonal elements give the eigenvalues of the original matrix $\mathbf{A}$, since

$$\mathbf{D} = \mathbf{V}^T \cdot \mathbf{A} \cdot \mathbf{V} \tag{11.1.21}$$

where

$$\mathbf{V} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdots \tag{11.1.22}$$

the $\mathbf{P}_i$'s being the successive Jacobi rotation matrices. The columns of $\mathbf{V}$ are the eigenvectors (since $\mathbf{A} \cdot \mathbf{V} = \mathbf{V} \cdot \mathbf{D}$). They can be computed by applying

$$\mathbf{V}' = \mathbf{V} \cdot \mathbf{P}_i \tag{11.1.23}$$

at each stage of calculation, where initially $\mathbf{V}$ is the identity matrix. In detail, equation (11.1.23) is

$$\begin{aligned} v'_{rs} &= v_{rs} \qquad (s \neq p,\ s \neq q) \\ v'_{rp} &= cv_{rp} - sv_{rq} \\ v'_{rq} &= sv_{rp} + cv_{rq} \end{aligned} \tag{11.1.24}$$

We rewrite these equations in terms of τ as in equations (11.1.16) and (11.1.17) to minimize roundoff.

The only remaining question is the strategy one should adopt for the order in which the elements are to be annihilated. Jacobi's original algorithm of 1846 searched the whole upper triangle at each stage and set the largest off-diagonal element to zero. This is a reasonable strategy for hand calculation, but it is prohibitive on a computer since the search alone makes each Jacobi rotation a process of order N^2 instead of N.

A better strategy for our purposes is the *cyclic Jacobi method*, where one annihilates elements in strict order. For example, one can simply proceed down the rows: $\mathbf{P}_{12}, \mathbf{P}_{13}, ..., \mathbf{P}_{1n}$; then $\mathbf{P}_{23}, \mathbf{P}_{24}$, etc. One can show that convergence is generally quadratic for both the original or the cyclic Jacobi methods, for nondegenerate eigenvalues. One such set of $n(n-1)/2$ Jacobi rotations is called a *sweep*.

The program below, based on the implementations in [1,2], uses two further refinements:

- In the first three sweeps, we carry out the pq rotation only if $|a_{pq}| > \epsilon$ for some threshold value

$$\epsilon = \frac{1}{5}\frac{S_0}{n^2} \tag{11.1.25}$$

  where S_0 is the sum of the off-diagonal moduli,

$$S_0 = \sum_{r<s} |a_{rs}| \tag{11.1.26}$$

- After four sweeps, if $|a_{pq}| \ll |a_{pp}|$ and $|a_{pq}| \ll |a_{qq}|$, we set $|a_{pq}| = 0$ and skip the rotation. The criterion used in the comparison is $|a_{pq}| < 10^{-(D+2)}|a_{pp}|$, where D is the number of significant decimal digits on the machine, and similarly for $|a_{qq}|$.

In the following routine the n×n symmetric matrix a(1:n,1:n) is stored in an np×np array. On output, the superdiagonal elements of a are destroyed, but the diagonal and subdiagonal are unchanged and give full information on the original symmetric matrix a. The parameter d is a vector of length np. On output, it returns the eigenvalues of a in its first n elements. During the computation, it contains the current diagonal of a. The matrix v outputs the normalized eigenvector belonging to d(k) in its kth column. The parameter nrot is the number of Jacobi rotations that were needed to achieve convergence.

Typical matrices require 6 to 10 sweeps to achieve convergence, or $3n^2$ to $5n^2$ Jacobi rotations. Each rotation requires of order $4n$ operations, each consisting of a multiply and an add, so the total labor is of order $12n^3$ to $20n^3$ operations. Calculation of the eigenvectors as well as the eigenvalues changes the operation count from $4n$ to $6n$ per rotation, which is only a 50 percent overhead.

```
SUBROUTINE jacobi(a,n,np,d,v,nrot)
INTEGER n,np,nrot,NMAX
REAL a(np,np),d(np),v(np,np)
PARAMETER (NMAX=500)
    Computes all eigenvalues and eigenvectors of a real symmetric matrix a, which is of size n
    by n, stored in a physical np by np array. On output, elements of a above the diagonal are
    destroyed. d returns the eigenvalues of a in its first n elements. v is a matrix with the same
    logical and physical dimensions as a, whose columns contain, on output, the normalized
    eigenvectors of a. nrot returns the number of Jacobi rotations that were required.
INTEGER i,ip,iq,j
REAL c,g,h,s,sm,t,tau,theta,tresh,b(NMAX),z(NMAX)
do 12 ip=1,n                            Initialize to the identity matrix.
    do 11 iq=1,n
        v(ip,iq)=0.
    enddo 11
    v(ip,ip)=1.
enddo 12
do 13 ip=1,n
    b(ip)=a(ip,ip)                      Initialize b and d to the diagonal of a.
    d(ip)=b(ip)
    z(ip)=0.                            This vector will accumulate terms of the form ta_pq
enddo 13                                    as in equation (11.1.14).
nrot=0
do 24 i=1,50
    sm=0.
    do 15 ip=1,n-1                      Sum off-diagonal elements.
        do 14 iq=ip+1,n
            sm=sm+abs(a(ip,iq))
        enddo 14
    enddo 15
    if(sm.eq.0.)return                  The normal return, which relies on quadratic conver-
    if(i.lt.4)then                          gence to machine underflow.
        tresh=0.2*sm/n**2               ...on the first three sweeps.
    else
        tresh=0.                        ...thereafter.
    endif
    do 22 ip=1,n-1
        do 21 iq=ip+1,n
            g=100.*abs(a(ip,iq))
              After four sweeps, skip the rotation if the off-diagonal element is small.
            if((i.gt.4).and.(abs(d(ip))+g.eq.abs(d(ip)))
*             .and.(abs(d(iq))+g.eq.abs(d(iq))))then
                a(ip,iq)=0.
            else if(abs(a(ip,iq)).gt.tresh)then
                h=d(iq)-d(ip)
                if(abs(h)+g.eq.abs(h))then
```

```
                t=a(ip,iq)/h      t = 1/(2θ)
            else
                theta=0.5*h/a(ip,iq)        Equation (11.1.10).
                t=1./(abs(theta)+sqrt(1.+theta**2))
                if(theta.lt.0.)t=-t
            endif
            c=1./sqrt(1+t**2)
            s=t*c
            tau=s/(1.+c)
            h=t*a(ip,iq)
            z(ip)=z(ip)-h
            z(iq)=z(iq)+h
            d(ip)=d(ip)-h
            d(iq)=d(iq)+h
            a(ip,iq)=0.
            do 16 j=1,ip-1          Case of rotations 1 ≤ j < p.
                g=a(j,ip)
                h=a(j,iq)
                a(j,ip)=g-s*(h+g*tau)
                a(j,iq)=h+s*(g-h*tau)
            enddo 16
            do 17 j=ip+1,iq-1       Case of rotations p < j < q.
                g=a(ip,j)
                h=a(j,iq)
                a(ip,j)=g-s*(h+g*tau)
                a(j,iq)=h+s*(g-h*tau)
            enddo 17
            do 18 j=iq+1,n          Case of rotations q < j ≤ n.
                g=a(ip,j)
                h=a(iq,j)
                a(ip,j)=g-s*(h+g*tau)
                a(iq,j)=h+s*(g-h*tau)
            enddo 18
            do 19 j=1,n
                g=v(j,ip)
                h=v(j,iq)
                v(j,ip)=g-s*(h+g*tau)
                v(j,iq)=h+s*(g-h*tau)
            enddo 19
            nrot=nrot+1
        endif
    enddo 21
  enddo 22
  do 23 ip=1,n
    b(ip)=b(ip)+z(ip)
    d(ip)=b(ip)                     Update d with the sum of ta_pq,
    z(ip)=0.                        and reinitialize z.
  enddo 23
enddo 24
pause 'too many iterations in jacobi'
return
END
```

Note that the above routine assumes that underflows are set to zero. On machines where this is not true, the program must be modified.

The eigenvalues are not ordered on output. If sorting is desired, the following routine can be invoked to reorder the output of `jacobi` or of later routines in this chapter. (The method, straight insertion, is N^2 rather than $N \log N$; but since you have just done an N^3 procedure to get the eigenvalues, you can afford yourself this little indulgence.)

```
SUBROUTINE eigsrt(d,v,n,np)
INTEGER n,np
REAL d(np),v(np,np)
    Given the eigenvalues d and eigenvectors v as output from jacobi (§11.1) or tqli (§11.3),
    this routine sorts the eigenvalues into descending order, and rearranges the columns of v
    correspondingly. The method is straight insertion.
INTEGER i,j,k
REAL p
do 13 i=1,n-1
    k=i
    p=d(i)
    do 11 j=i+1,n
        if(d(j).ge.p)then
            k=j
            p=d(j)
        endif
    enddo 11
    if(k.ne.i)then
        d(k)=d(i)
        d(i)=p
        do 12 j=1,n
            p=v(j,i)
            v(j,i)=v(j,k)
            v(j,k)=p
        enddo 12
    endif
enddo 13
return
END
```

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.4.

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

11.2 Reduction of a Symmetric Matrix to Tridiagonal Form: Givens and Householder Reductions

As already mentioned, the optimum strategy for finding eigenvalues and eigenvectors is, first, to reduce the matrix to a simple form, only then beginning an iterative procedure. For symmetric matrices, the preferred simple form is tridiagonal. The *Givens reduction* is a modification of the Jacobi method. Instead of trying to reduce the matrix all the way to diagonal form, we are content to stop when the matrix is tridiagonal. This allows the procedure to be carried out *in a finite number of steps*, unlike the Jacobi method, which requires iteration to convergence.

```
SUBROUTINE eigsrt(d,v,n,np)
INTEGER n,np
REAL d(np),v(np,np)
    Given the eigenvalues d and eigenvectors v as output from jacobi (§11.1) or tqli (§11.3),
    this routine sorts the eigenvalues into descending order, and rearranges the columns of v
    correspondingly. The method is straight insertion.
INTEGER i,j,k
REAL p
do 13 i=1,n-1
    k=i
    p=d(i)
    do 11 j=i+1,n
        if(d(j).ge.p)then
            k=j
            p=d(j)
        endif
    enddo 11
    if(k.ne.i)then
        d(k)=d(i)
        d(i)=p
        do 12 j=1,n
            p=v(j,i)
            v(j,i)=v(j,k)
            v(j,k)=p
        enddo 12
    endif
enddo 13
return
END
```

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.4.

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

11.2 Reduction of a Symmetric Matrix to Tridiagonal Form: Givens and Householder Reductions

As already mentioned, the optimum strategy for finding eigenvalues and eigenvectors is, first, to reduce the matrix to a simple form, only then beginning an iterative procedure. For symmetric matrices, the preferred simple form is tridiagonal. The *Givens reduction* is a modification of the Jacobi method. Instead of trying to reduce the matrix all the way to diagonal form, we are content to stop when the matrix is tridiagonal. This allows the procedure to be carried out *in a finite number of steps*, unlike the Jacobi method, which requires iteration to convergence.

Givens Method

For the Givens method, we choose the rotation angle in equation (11.1.1) so as to zero an element that is *not* at one of the four "corners," i.e., not a_{pp}, a_{pq}, or a_{qq} in equation (11.1.3). Specifically, we first choose $\mathbf{P}_{23}$ to annihilate a_{31} (and, by symmetry, a_{13}). Then we choose $\mathbf{P}_{24}$ to annihilate a_{41}. In general, we choose the sequence

$$\mathbf{P}_{23}, \mathbf{P}_{24}, \ldots, \mathbf{P}_{2n}; \mathbf{P}_{34}, \ldots, \mathbf{P}_{3n}; \ldots; \mathbf{P}_{n-1,n}$$

where $\mathbf{P}_{jk}$ annihilates $a_{k,j-1}$. The method works because elements such as a'_{rp} and a'_{rq}, with $r \neq p$ $r \neq q$, are linear combinations of the old quantities a_{rp} and a_{rq}, by equation (11.1.4). Thus, if a_{rp} and a_{rq} have already been set to zero, they remain zero as the reduction proceeds. Evidently, of order $n^2/2$ rotations are required, and the number of multiplications in a straightforward implementation is of order $4n^3/3$, not counting those for keeping track of the product of the transformation matrices, required for the eigenvectors.

The Householder method, to be discussed next, is just as stable as the Givens reduction and it is a factor of 2 more efficient, so the Givens method is not generally used. Recent work (see [1]) has shown that the Givens reduction can be reformulated to reduce the number of operations by a factor of 2, and also avoid the necessity of taking square roots. This appears to make the algorithm competitive with the Householder reduction. However, this "fast Givens" reduction has to be monitored to avoid overflows, and the variables have to be periodically rescaled. There does not seem to be any compelling reason to prefer the Givens reduction over the Householder method.

Householder Method

The Householder algorithm reduces an $n \times n$ symmetric matrix $\mathbf{A}$ to tridiagonal form by $n-2$ orthogonal transformations. Each transformation annihilates the required part of a whole column and whole corresponding row. The basic ingredient is a Householder matrix $\mathbf{P}$, which has the form

$$\mathbf{P} = \mathbf{1} - 2\mathbf{w} \cdot \mathbf{w}^T \tag{11.2.1}$$

where $\mathbf{w}$ is a real vector with $|\mathbf{w}|^2 = 1$. (In the present notation, the *outer* or matrix product of two vectors, $\mathbf{a}$ and $\mathbf{b}$ is written $\mathbf{a} \cdot \mathbf{b}^T$, while the *inner* or scalar product of the vectors is written as $\mathbf{a}^T \cdot \mathbf{b}$.) The matrix $\mathbf{P}$ is orthogonal, because

$$\begin{aligned} \mathbf{P}^2 &= (\mathbf{1} - 2\mathbf{w} \cdot \mathbf{w}^T) \cdot (\mathbf{1} - 2\mathbf{w} \cdot \mathbf{w}^T) \\ &= \mathbf{1} - 4\mathbf{w} \cdot \mathbf{w}^T + 4\mathbf{w} \cdot (\mathbf{w}^T \cdot \mathbf{w}) \cdot \mathbf{w}^T \\ &= \mathbf{1} \end{aligned} \tag{11.2.2}$$

Therefore $\mathbf{P} = \mathbf{P}^{-1}$. But $\mathbf{P}^T = \mathbf{P}$, and so $\mathbf{P}^T = \mathbf{P}^{-1}$, proving orthogonality.

Rewrite $\mathbf{P}$ as

$$\mathbf{P} = \mathbf{1} - \frac{\mathbf{u} \cdot \mathbf{u}^T}{H} \tag{11.2.3}$$

where the scalar H is

$$H \equiv \frac{1}{2} |\mathbf{u}|^2 \tag{11.2.4}$$

and $\mathbf{u}$ can now be any vector. Suppose $\mathbf{x}$ is the vector composed of the first column of $\mathbf{A}$. Choose

$$\mathbf{u} = \mathbf{x} \mp |\mathbf{x}| \mathbf{e}_1 \tag{11.2.5}$$

where $\mathbf{e}_1$ is the unit vector $[1, 0, \ldots, 0]^T$, and the choice of signs will be made later. Then

$$\begin{aligned} \mathbf{P} \cdot \mathbf{x} &= \mathbf{x} - \frac{\mathbf{u}}{H} \cdot (\mathbf{x} \mp |\mathbf{x}| \mathbf{e}_1)^T \cdot \mathbf{x} \\ &= \mathbf{x} - \frac{2\mathbf{u} \cdot (|\mathbf{x}|^2 \mp |\mathbf{x}| x_1)}{2|\mathbf{x}|^2 \mp 2|\mathbf{x}| x_1} \\ &= \mathbf{x} - \mathbf{u} \\ &= \pm |\mathbf{x}| \mathbf{e}_1 \end{aligned} \tag{11.2.6}$$

This shows that the Householder matrix $\mathbf{P}$ acts on a given vector $\mathbf{x}$ to zero all its elements except the first one.

To reduce a symmetric matrix $\mathbf{A}$ to tridiagonal form, we choose the vector $\mathbf{x}$ for the first Householder matrix to be the lower $n-1$ elements of the first column. Then the lower $n-2$ elements will be zeroed:

$$\begin{aligned} \mathbf{P}_1 \cdot \mathbf{A} &= \left[\begin{array}{c|ccc} 1 & 0 & 0 & \cdots & 0 \\ \hline 0 & & & & \\ 0 & & & & \\ \vdots & & {}^{(n-1)}\mathbf{P}_1 & & \\ 0 & & & & \end{array}\right] \cdot \left[\begin{array}{c|ccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline a_{21} & & & & \\ a_{31} & & & & \\ \vdots & & \text{irrelevant} & & \\ a_{n1} & & & & \end{array}\right] \\ &= \left[\begin{array}{c|ccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline k & & & & \\ 0 & & & & \\ \vdots & & \text{irrelevant} & & \\ 0 & & & & \end{array}\right] \end{aligned} \tag{11.2.7}$$

Here we have written the matrices in partitioned form, with ${}^{(n-1)}\mathbf{P}$ denoting a Householder matrix with dimensions $(n-1) \times (n-1)$. The quantity k is simply plus or minus the magnitude of the vector $[a_{21}, \ldots, a_{n1}]^T$.

The complete orthogonal transformation is now

$$\mathbf{A}' = \mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P} = \left[\begin{array}{c|ccc} a_{11} & k & 0 & \cdots & 0 \\ \hline k & & & & \\ 0 & & & & \\ \vdots & & \text{irrelevant} & & \\ 0 & & & & \end{array}\right] \tag{11.2.8}$$

We have used the fact that $\mathbf{P}^T = \mathbf{P}$.

Now choose the vector $\mathbf{x}$ for the second Householder matrix to be the bottom $n-2$ elements of the second column, and from it construct

$$\mathbf{P}_2 \equiv \left[\begin{array}{cc|ccc} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & {}^{(n-2)}\mathbf{P}_2 & \\ 0 & 0 & & & \end{array}\right] \tag{11.2.9}$$

The identity block in the upper left corner insures that the tridiagonalization achieved in the first step will not be spoiled by this one, while the $(n-2)$-dimensional Householder matrix ${}^{(n-2)}\mathbf{P}_2$ creates one additional row and column of the tridiagonal output. Clearly, a sequence of $n-2$ such transformations will reduce the matrix $\mathbf{A}$ to tridiagonal form.

Instead of actually carrying out the matrix multiplications in $\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}$, we compute a vector

$$\mathbf{p} \equiv \frac{\mathbf{A} \cdot \mathbf{u}}{H} \tag{11.2.10}$$

Then

$$\mathbf{A} \cdot \mathbf{P} = \mathbf{A} \cdot (1 - \frac{\mathbf{u} \cdot \mathbf{u}^T}{H}) = \mathbf{A} - \mathbf{p} \cdot \mathbf{u}^T$$
$$\mathbf{A}' = \mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P} = \mathbf{A} - \mathbf{p} \cdot \mathbf{u}^T - \mathbf{u} \cdot \mathbf{p}^T + 2K\mathbf{u} \cdot \mathbf{u}^T$$

where the scalar K is defined by

$$K = \frac{\mathbf{u}^T \cdot \mathbf{p}}{2H} \tag{11.2.11}$$

If we write

$$\mathbf{q} \equiv \mathbf{p} - K\mathbf{u} \tag{11.2.12}$$

then we have

$$\mathbf{A}' = \mathbf{A} - \mathbf{q} \cdot \mathbf{u}^T - \mathbf{u} \cdot \mathbf{q}^T \tag{11.2.13}$$

This is the computationally useful formula.

Following [2], the routine for Householder reduction given below actually starts in the nth column of $\mathbf{A}$, not the first as in the explanation above. In detail, the equations are as follows: At stage m ($m = 1, 2, \ldots, n-2$) the vector $\mathbf{u}$ has the form

$$\mathbf{u}^T = [a_{i1}, a_{i2}, \ldots, a_{i,i-2},\ a_{i,i-1} \pm \sqrt{\sigma},\ 0, \ldots, 0] \tag{11.2.14}$$

Here

$$i \equiv n - m + 1 = n, n - 1, \ldots, 3 \tag{11.2.15}$$

and the quantity σ ($|x|^2$ in our earlier notation) is

$$\sigma = (a_{i1})^2 + \cdots + (a_{i,i-1})^2 \tag{11.2.16}$$

We choose the sign of σ in (11.2.14) to be the same as the sign of $a_{i,i-1}$ to lessen roundoff error.

Variables are thus computed in the following order: $\sigma, \mathbf{u}, H, \mathbf{p}, K, \mathbf{q}, \mathbf{A}'$. At any stage m, $\mathbf{A}$ is tridiagonal in its last $m-1$ rows and columns.

If the eigenvectors of the final tridiagonal matrix are found (for example, by the routine in the next section), then the eigenvectors of $\mathbf{A}$ can be obtained by applying the accumulated transformation

$$\mathbf{Q} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdots \mathbf{P}_{n-2} \tag{11.2.17}$$

to those eigenvectors. We therefore form $\mathbf{Q}$ by recursion after all the $\mathbf{P}$'s have been determined:

$$\begin{aligned} \mathbf{Q}_{n-2} &= \mathbf{P}_{n-2} \\ \mathbf{Q}_j &= \mathbf{P}_j \cdot \mathbf{Q}_{j+1}, \qquad j = n-3, \ldots, 1 \\ \mathbf{Q} &= \mathbf{Q}_1 \end{aligned} \tag{11.2.18}$$

The input parameters for the routine below are the `n`×`n` real, symmetric matrix `a`, stored in an `np`×`np` array. On output, `a` contains the elements of the orthogonal matrix `q`. The vector `d` returns the diagonal elements of the tridiagonal matrix $\mathbf{A}'$, while the vector `e` returns the off-diagonal elements in its components `2` through `n`, with `e(1)=0`. Note that since `a` is overwritten, you should copy it before calling the routine, if it is required for subsequent computations.

No extra storage arrays are needed for the intermediate results. At stage m, the vectors $\mathbf{p}$ and $\mathbf{q}$ are nonzero only in elements $1, \ldots, i$ (recall that $i = n - m + 1$), while $\mathbf{u}$ is nonzero only in elements $1, \ldots, i-1$. The elements of the vector `e` are being determined in the order $n, n-1, \ldots$, so we can store $\mathbf{p}$ in the elements of `e` not already determined. The vector $\mathbf{q}$ can overwrite $\mathbf{p}$ once $\mathbf{p}$ is no longer needed. We store $\mathbf{u}$ in the ith row of `a` and $\mathbf{u}/H$ in the ith column of `a`. Once the reduction is complete, we compute the matrices $\mathbf{Q}_j$ using the quantities $\mathbf{u}$ and $\mathbf{u}/H$ that have been stored in `a`. Since $\mathbf{Q}_j$ is an identity matrix in the last $n - j + 1$ rows and columns, we only need compute its elements up to row and column $n - j$. These can overwrite the $\mathbf{u}$'s and $\mathbf{u}/H$'s in the corresponding rows and columns of `a`, which are no longer required for subsequent $\mathbf{Q}$'s.

The routine `tred2`, given below, includes one further refinement. If the quantity σ is zero or "small" at any stage, one can skip the corresponding transformation. A simple criterion, such as

$$\sigma < \frac{\text{smallest positive number representable on machine}}{\text{machine precision}}$$

would be fine most of the time. A more careful criterion is actually used. Define the quantity

$$\epsilon = \sum_{k=1}^{i-1} |a_{ik}| \tag{11.2.19}$$

If $\epsilon = 0$ to machine precision, we skip the transformation. Otherwise we redefine

$$a_{ik} \quad \text{becomes} \quad a_{ik}/\epsilon \tag{11.2.20}$$

and use the scaled variables for the transformation. (A Householder transformation depends only on the ratios of the elements.)

Note that when dealing with a matrix whose elements vary over many orders of magnitude, it is important that the matrix be permuted, insofar as possible, so that the smaller elements are in the top left-hand corner. This is because the reduction is performed starting from the bottom right-hand corner, and a mixture of small and large elements there can lead to considerable rounding errors.

The routine `tred2` is designed for use with the routine `tqli` of the next section. `tqli` finds the eigenvalues and eigenvectors of a symmetric, tridiagonal matrix. The combination of `tred2` and `tqli` is the most efficient known technique for finding all the eigenvalues and eigenvectors (or just all the eigenvalues) of a real, symmetric matrix.

In the listing below, the statements indicated by comments are required only for subsequent computation of eigenvectors. If only eigenvalues are required, omission of the commented statements speeds up the execution time of `tred2` by a factor of 2 for large n. In the limit of large n, the operation count of the Householder reduction is $2n^3/3$ for eigenvalues only, and $4n^3/3$ for both eigenvalues and eigenvectors.

```
SUBROUTINE tred2(a,n,np,d,e)
INTEGER n,np
REAL a(np,np),d(np),e(np)
    Householder reduction of a real, symmetric, n by n matrix a, stored in an np by np physical
    array. On output, a is replaced by the orthogonal matrix Q effecting the transformation. d
    returns the diagonal elements of the tridiagonal matrix, and e the off-diagonal elements,
    with e(1)=0. Several statements, as noted in comments, can be omitted if only eigenvalues
    are to be found, in which case a contains no useful information on output. Otherwise they
    are to be included.
INTEGER i,j,k,l
REAL f,g,h,hh,scale
do 18 i=n,2,-1
    l=i-1
    h=0.
    scale=0.
    if(l.gt.1)then
        do 11 k=1,l
            scale=scale+abs(a(i,k))
        enddo 11
        if(scale.eq.0.)then                Skip transformation.
            e(i)=a(i,l)
        else
            do 12 k=1,l
                a(i,k)=a(i,k)/scale        Use scaled a's for transformation.
                h=h+a(i,k)**2              Form σ in h.
            enddo 12
            f=a(i,l)
```

```
            g=-sign(sqrt(h),f)
            e(i)=scale*g
            h=h-f*g                     Now h is equation (11.2.4).
            a(i,l)=f-g                  Store u in the ith row of a.
            f=0.
            do 15 j=1,l
C   Omit following line if finding only eigenvalues
                a(j,i)=a(i,j)/h         Store u/H in ith column of a.
                g=0.                    Form an element of A · u in g.
                do 13 k=1,j
                    g=g+a(j,k)*a(i,k)
                enddo 13
                do 14 k=j+1,l
                    g=g+a(k,j)*a(i,k)
                enddo 14
                e(j)=g/h                Form element of p in temporarily unused
                f=f+e(j)*a(i,j)             element of e.
            enddo 15
            hh=f/(h+h)                  Form K, equation (11.2.11).
            do 17 j=1,l                 Form q and store in e overwriting p.
                f=a(i,j)
                g=e(j)-hh*f
                e(j)=g
                do 16 k=1,j             Reduce a, equation (11.2.13).
                    a(j,k)=a(j,k)-f*e(k)-g*a(i,k)
                enddo 16
            enddo 17
        endif
    else
        e(i)=a(i,l)
    endif
    d(i)=h
enddo 18
C   Omit following line if finding only eigenvalues.
d(1)=0.
e(1)=0.
do 24 i=1,n                             Begin accumulation of transformation matrices.
C   Delete lines from here ...
    l=i-1
    if(d(i).ne.0.)then                  This block skipped when i=1.
        do 22 j=1,l
            g=0.
            do 19 k=1,l                 Use u and u/H stored in a to form P · Q.
                g=g+a(i,k)*a(k,j)
            enddo 19
            do 21 k=1,l
                a(k,j)=a(k,j)-g*a(k,i)
            enddo 21
        enddo 22
    endif
C   ... to here when finding only eigenvalues.
    d(i)=a(i,i)                         This statement remains.
C   Also delete lines from here ...
    a(i,i)=1.                           Reset row and column of a to identity matrix for
    do 23 j=1,l                             next iteration.
        a(i,j)=0.
        a(j,i)=0.
    enddo 23
C   ... to here when finding only eigenvalues.
enddo 24
return
END
```

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.1. [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

11.3 Eigenvalues and Eigenvectors of a Tridiagonal Matrix

Evaluation of the Characteristic Polynomial

Once our original, real, symmetric matrix has been reduced to tridiagonal form, one possible way to determine its eigenvalues is to find the roots of the characteristic polynomial $p_n(\lambda)$ directly. The characteristic polynomial of a tridiagonal matrix can be evaluated for any trial value of λ by an efficient recursion relation (see [1], for example). The polynomials of lower degree produced during the recurrence form a Sturmian sequence that can be used to localize the eigenvalues to intervals on the real axis. A root-finding method such as bisection or Newton's method can then be employed to refine the intervals. The corresponding eigenvectors can then be found by inverse iteration (see §11.7).

Procedures based on these ideas can be found in [2,3]. If, however, more than a small fraction of all the eigenvalues and eigenvectors are required, then the factorization method next considered is much more efficient.

The QR and QL Algorithms

The basic idea behind the QR algorithm is that any real matrix can be decomposed in the form

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{11.3.1}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper triangular. For a general matrix, the decomposition is constructed by applying Householder transformations to annihilate successive columns of $\mathbf{A}$ below the diagonal (see §2.10).

Now consider the matrix formed by writing the factors in (11.3.1) in the opposite order:

$$\mathbf{A}' = \mathbf{R} \cdot \mathbf{Q} \tag{11.3.2}$$

Since $\mathbf{Q}$ is orthogonal, equation (11.3.1) gives $\mathbf{R} = \mathbf{Q}^T \cdot \mathbf{A}$. Thus equation (11.3.2) becomes

$$\mathbf{A}' = \mathbf{Q}^T \cdot \mathbf{A} \cdot \mathbf{Q} \tag{11.3.3}$$

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §5.1. [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

11.3 Eigenvalues and Eigenvectors of a Tridiagonal Matrix

Evaluation of the Characteristic Polynomial

Once our original, real, symmetric matrix has been reduced to tridiagonal form, one possible way to determine its eigenvalues is to find the roots of the characteristic polynomial $p_n(\lambda)$ directly. The characteristic polynomial of a tridiagonal matrix can be evaluated for any trial value of λ by an efficient recursion relation (see [1], for example). The polynomials of lower degree produced during the recurrence form a Sturmian sequence that can be used to localize the eigenvalues to intervals on the real axis. A root-finding method such as bisection or Newton's method can then be employed to refine the intervals. The corresponding eigenvectors can then be found by inverse iteration (see §11.7).

Procedures based on these ideas can be found in [2,3]. If, however, more than a small fraction of all the eigenvalues and eigenvectors are required, then the factorization method next considered is much more efficient.

The QR and QL Algorithms

The basic idea behind the QR algorithm is that any real matrix can be decomposed in the form

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{11.3.1}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper triangular. For a general matrix, the decomposition is constructed by applying Householder transformations to annihilate successive columns of $\mathbf{A}$ below the diagonal (see §2.10).

Now consider the matrix formed by writing the factors in (11.3.1) in the opposite order:

$$\mathbf{A}' = \mathbf{R} \cdot \mathbf{Q} \tag{11.3.2}$$

Since $\mathbf{Q}$ is orthogonal, equation (11.3.1) gives $\mathbf{R} = \mathbf{Q}^T \cdot \mathbf{A}$. Thus equation (11.3.2) becomes

$$\mathbf{A}' = \mathbf{Q}^T \cdot \mathbf{A} \cdot \mathbf{Q} \tag{11.3.3}$$

We see that $\mathbf{A}'$ is an orthogonal transformation of $\mathbf{A}$.

You can verify that a QR transformation preserves the following properties of a matrix: symmetry, tridiagonal form, and Hessenberg form (to be defined in §11.5).

There is nothing special about choosing one of the factors of $\mathbf{A}$ to be upper triangular; one could equally well make it lower triangular. This is called the QL algorithm, since

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{L} \tag{11.3.4}$$

where $\mathbf{L}$ is lower triangular. (The standard, but confusing, nomenclature R and L stands for whether the *right* or *left* of the matrix is nonzero.)

Recall that in the Householder reduction to tridiagonal form in §11.2, we started in the nth (last) column of the original matrix. To minimize roundoff, we then exhorted you to put the biggest elements of the matrix in the lower right-hand corner, if you can. If we now wish to diagonalize the resulting tridiagonal matrix, the QL algorithm will have smaller roundoff than the QR algorithm, so we shall use QL henceforth.

The QL algorithm consists of a *sequence* of orthogonal transformations:

$$\begin{aligned} \mathbf{A}_s &= \mathbf{Q}_s \cdot \mathbf{L}_s \\ \mathbf{A}_{s+1} &= \mathbf{L}_s \cdot \mathbf{Q}_s \qquad (= \mathbf{Q}_s^T \cdot \mathbf{A}_s \cdot \mathbf{Q}_s) \end{aligned} \tag{11.3.5}$$

The following (nonobvious!) theorem is the basis of the algorithm for a general matrix $\mathbf{A}$: (i) If $\mathbf{A}$ has eigenvalues of different absolute value $|\lambda_i|$, then $\mathbf{A}_s \to$ [lower triangular form] as $s \to \infty$. The eigenvalues appear on the diagonal in increasing order of absolute magnitude. (ii) If $\mathbf{A}$ has an eigenvalue $|\lambda_i|$ of multiplicity p, $\mathbf{A}_s \to$ [lower triangular form] as $s \to \infty$, except for a diagonal block matrix of order p, whose eigenvalues $\to \lambda_i$. The proof of this theorem is fairly lengthy; see, for example, [4].

The workload in the QL algorithm is $O(n^3)$ per iteration for a general matrix, which is prohibitive. However, the workload is only $O(n)$ per iteration for a tridiagonal matrix and $O(n^2)$ for a Hessenberg matrix, which makes it highly efficient on these forms.

In this section we are concerned only with the case where $\mathbf{A}$ is a real, symmetric, tridiagonal matrix. All the eigenvalues λ_i are thus real. According to the theorem, if any λ_i has a multiplicity p, then there must be at least $p-1$ zeros on the sub- and superdiagonal. Thus the matrix can be split into submatrices that can be diagonalized separately, and the complication of diagonal blocks that can arise in the general case is irrelevant.

In the proof of the theorem quoted above, one finds that in general a superdiagonal element converges to zero like

$$a_{ij}^{(s)} \sim \left(\frac{\lambda_i}{\lambda_j}\right)^s \tag{11.3.6}$$

Although $\lambda_i < \lambda_j$, convergence can be slow if λ_i is close to λ_j. Convergence can be accelerated by the technique of *shifting*: If k is any constant, then $\mathbf{A} - k\mathbf{1}$ has

eigenvalues $\lambda_i - k$. If we decompose

$$\mathbf{A}_s - k_s\mathbf{1} = \mathbf{Q}_s \cdot \mathbf{L}_s \tag{11.3.7}$$

so that

$$\begin{aligned}\mathbf{A}_{s+1} &= \mathbf{L}_s \cdot \mathbf{Q}_s + k_s\mathbf{1} \\ &= \mathbf{Q}_s^T \cdot \mathbf{A}_s \cdot \mathbf{Q}_s\end{aligned} \tag{11.3.8}$$

then the convergence is determined by the ratio

$$\frac{\lambda_i - k_s}{\lambda_j - k_s} \tag{11.3.9}$$

The idea is to choose the shift k_s at each stage to maximize the rate of convergence. A good choice for the shift initially would be k_s close to λ_1, the smallest eigenvalue. Then the first row of off-diagonal elements would tend rapidly to zero. However, λ_1 is not usually known *a priori*. A very effective strategy in practice (although there is no proof that it is optimal) is to compute the eigenvalues of the leading 2×2 diagonal submatrix of $\mathbf{A}$. Then set k_s equal to the eigenvalue closer to a_{11}.

More generally, suppose you have already found $r-1$ eigenvalues of $\mathbf{A}$. Then you can *deflate* the matrix by crossing out the first $r-1$ rows and columns, leaving

$$\mathbf{A} = \begin{bmatrix} 0 & & & \cdots & \cdots & & & 0 \\ & \cdots & & & & & & \\ & & 0 & & & & & \\ \vdots & & & d_r & e_r & & & \vdots \\ \vdots & & & e_r & d_{r+1} & & & \\ & & & & & \cdots & & 0 \\ & & & & & & d_{n-1} & e_{n-1} \\ 0 & & & \cdots & & 0 & e_{n-1} & d_n \end{bmatrix} \tag{11.3.10}$$

Choose k_s equal to the eigenvalue of the leading 2×2 submatrix that is closer to d_r. One can show that the convergence of the algorithm with this strategy is generally cubic (and at worst quadratic for degenerate eigenvalues). This rapid convergence is what makes the algorithm so attractive.

Note that with shifting, the eigenvalues no longer necessarily appear on the diagonal in order of increasing absolute magnitude. The routine `eigsrt` (§11.1) can be used if required.

As we mentioned earlier, the QL decomposition of a general matrix is effected by a sequence of Householder transformations. For a tridiagonal matrix, however, it is more efficient to use plane rotations $\mathbf{P}_{pq}$. One uses the sequence $\mathbf{P}_{12}, \mathbf{P}_{23}, \ldots, \mathbf{P}_{n-1,n}$ to annihilate the elements $a_{12}, a_{23}, \ldots, a_{n-1,n}$. By symmetry, the subdiagonal elements $a_{21}, a_{32}, \ldots, a_{n,n-1}$ will be annihilated too. Thus each $\mathbf{Q}_s$ is a product of plane rotations:

$$\mathbf{Q}_s^T = \mathbf{P}_1^{(s)} \cdot \mathbf{P}_2^{(s)} \cdots \mathbf{P}_{n-1}^{(s)} \tag{11.3.11}$$

where $\mathbf{P}_i$ annihilates $a_{i,i+1}$. Note that it is $\mathbf{Q}^T$ in equation (11.3.11), not $\mathbf{Q}$, because we defined $\mathbf{L} = \mathbf{Q}^T \cdot \mathbf{A}$.

QL Algorithm with Implicit Shifts

The algorithm as described so far can be very successful. However, when the elements of $\mathbf{A}$ differ widely in order of magnitude, subtracting a large k_s from the diagonal elements can lead to loss of accuracy for the small eigenvalues. This difficulty is avoided by the QL algorithm with *implicit shifts*. The implicit QL algorithm is mathematically equivalent to the original QL algorithm, but the computation does not require $k_s\mathbf{1}$ to be actually subtracted from $\mathbf{A}$.

The algorithm is based on the following lemma: If $\mathbf{A}$ is a symmetric nonsingular matrix and $\mathbf{B} = \mathbf{Q}^T \cdot \mathbf{A} \cdot \mathbf{Q}$, where $\mathbf{Q}$ is orthogonal and $\mathbf{B}$ is tridiagonal with positive off-diagonal elements, then $\mathbf{Q}$ and $\mathbf{B}$ are fully determined when the last row of $\mathbf{Q}^T$ is specified. Proof: Let $\mathbf{q}_i^T$ denote the ith row vector of the matrix $\mathbf{Q}^T$. Then $\mathbf{q}_i$ is the ith column vector of the matrix $\mathbf{Q}$. The relation $\mathbf{B} \cdot \mathbf{Q}^T = \mathbf{Q}^T \cdot \mathbf{A}$ can be written

$$\begin{bmatrix} \beta_1 & \gamma_1 & & & & \\ \alpha_2 & \beta_2 & \gamma_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} & \\ & & & \alpha_n & \beta_n & \end{bmatrix} \cdot \begin{bmatrix} \mathbf{q}_1^T \\ \mathbf{q}_2^T \\ \vdots \\ \mathbf{q}_{n-1}^T \\ \mathbf{q}_n^T \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1^T \\ \mathbf{q}_2^T \\ \vdots \\ \mathbf{q}_{n-1}^T \\ \mathbf{q}_n^T \end{bmatrix} \cdot \mathbf{A} \tag{11.3.12}$$

The nth row of this matrix equation is

$$\alpha_n \mathbf{q}_{n-1}^T + \beta_n \mathbf{q}_n^T = \mathbf{q}_n^T \cdot \mathbf{A} \tag{11.3.13}$$

Since $\mathbf{Q}$ is orthogonal,

$$\mathbf{q}_n^T \cdot \mathbf{q}_m = \delta_{nm} \tag{11.3.14}$$

Thus if we postmultiply equation (11.3.13) by $\mathbf{q}_n$, we find

$$\beta_n = \mathbf{q}_n^T \cdot \mathbf{A} \cdot \mathbf{q}_n \tag{11.3.15}$$

which is known since $\mathbf{q}_n$ is known. Then equation (11.3.13) gives

$$\alpha_n \mathbf{q}_{n-1}^T = \mathbf{z}_{n-1}^T \tag{11.3.16}$$

where

$$\mathbf{z}_{n-1}^T \equiv \mathbf{q}_n^T \cdot \mathbf{A} - \beta_n \mathbf{q}_n^T \tag{11.3.17}$$

is known. Therefore

$$\alpha_n^2 = \mathbf{z}_{n-1}^T \mathbf{z}_{n-1}, \tag{11.3.18}$$

or

$$\alpha_n = |\mathbf{z}_{n-1}| \tag{11.3.19}$$

and

$$\mathbf{q}_{n-1}^T = \mathbf{z}_{n-1}^T / \alpha_n \tag{11.3.20}$$

(where α_n is nonzero by hypothesis). Similarly, one can show by induction that if we know $\mathbf{q}_n, \mathbf{q}_{n-1}, \ldots, \mathbf{q}_{n-j}$ and the α's, β's, and γ's up to level $n-j$, one can determine the quantities at level $n-(j+1)$.

To apply the lemma in practice, suppose one can somehow find a tridiagonal matrix $\overline{\mathbf{A}}_{s+1}$ such that

$$\overline{\mathbf{A}}_{s+1} = \overline{\mathbf{Q}}_s^T \cdot \overline{\mathbf{A}}_s \cdot \overline{\mathbf{Q}}_s \tag{11.3.21}$$

where $\overline{\mathbf{Q}}_s^T$ is orthogonal and has the same last row as $\mathbf{Q}_s^T$ in the original QL algorithm. Then $\overline{\mathbf{Q}}_s = \mathbf{Q}_s$ and $\overline{\mathbf{A}}_{s+1} = \mathbf{A}_{s+1}$.

Now, in the original algorithm, from equation (11.3.11) we see that the last row of $\mathbf{Q}_s^T$ is the same as the last row of $\mathbf{P}_{n-1}^{(s)}$. But recall that $\mathbf{P}_{n-1}^{(s)}$ is a plane rotation designed to annihilate the $(n-1,n)$ element of $\mathbf{A}_s - k_s\mathbf{1}$. A simple calculation using the expression (11.1.1) shows that it has parameters

$$c = \frac{d_n - k_s}{\sqrt{e_n^2 + (d_n - k_s)^2}} \quad , \quad s = \frac{-e_{n-1}}{\sqrt{e_n^2 + (d_n - k_s)^2}} \tag{11.3.22}$$

The matrix $\mathbf{P}_{n-1}^{(s)} \cdot \mathbf{A}_s \cdot \mathbf{P}_{n-1}^{(s)T}$ is tridiagonal with 2 extra elements:

$$\begin{bmatrix} \cdots & & & & & & \\ & \times & \times & \times & & & \\ & & \times & \times & \times & \mathbf{x} & \\ & & & \times & \times & \times & \\ & & & \mathbf{x} & \times & \times & \end{bmatrix} \tag{11.3.23}$$

We must now reduce this to tridiagonal form with an orthogonal matrix whose last row is $[0,0,\ldots,0,1]$ so that the last row of $\overline{\mathbf{Q}}_s^T$ will stay equal to $\mathbf{P}_{n-1}^{(s)}$. This can be done by a sequence of Householder or Givens transformations. For the special form of the matrix (11.3.23), Givens is better. We rotate in the plane $(n-2,n-1)$ to annihilate the $(n-2,n)$ element. [By symmetry, the $(n,n-2)$ element will also be zeroed.] This leaves us with tridiagonal form except for extra elements $(n-3,n-1)$ and $(n-1,n-3)$. We annihilate these with a rotation in the $(n-3,n-2)$ plane, and so on. Thus a sequence of $n-2$ Givens rotations is required. The result is that

$$\mathbf{Q}_s^T = \overline{\mathbf{Q}}_s^T = \overline{\mathbf{P}}_1^{(s)} \cdot \overline{\mathbf{P}}_2^{(s)} \cdots \overline{\mathbf{P}}_{n-2}^{(s)} \cdot \mathbf{P}_{n-1}^{(s)} \tag{11.3.24}$$

where the $\overline{\mathbf{P}}$'s are the Givens rotations and $\mathbf{P}_{n-1}$ is the same plane rotation as in the original algorithm. Then equation (11.3.21) gives the next iterate of $\mathbf{A}$. Note that the shift k_s enters implicitly through the parameters (11.3.22).

The following routine `tqli` ("Tridiagonal QL Implicit"), based algorithmically on the implementations in [2,3], works extremely well in practice. The number of iterations for the first few eigenvalues might be 4 or 5, say, but meanwhile the off-diagonal elements in the lower right-hand corner have been reduced too. The later eigenvalues are liberated with very little work. The average number of iterations per eigenvalue is typically $1.3 - 1.6$. The operation count per iteration is $O(n)$, with a fairly large effective coefficient, say, $\sim 20n$. The total operation count for the diagonalization is then $\sim 20n \times (1.3 - 1.6)n \sim 30n^2$. If the eigenvectors are required, the statements indicated by comments are included and there is an additional, much larger, workload of about $3n^3$ operations.

```
      SUBROUTINE tqli(d,e,n,np,z)
      INTEGER n,np
      REAL d(np),e(np),z(np,np)
C     USES pythag
```

QL algorithm with implicit shifts, to determine the eigenvalues and eigenvectors of a real, symmetric, tridiagonal matrix, or of a real, symmetric matrix previously reduced by `tred2` §11.2. `d` is a vector of length `np`. On input, its first `n` elements are the diagonal elements of the tridiagonal matrix. On output, it returns the eigenvalues. The vector `e` inputs the subdiagonal elements of the tridiagonal matrix, with `e(1)` arbitrary. On output `e` is destroyed. When finding only the eigenvalues, several lines may be omitted, as noted in the comments. If the eigenvectors of a tridiagonal matrix are desired, the matrix `z` (`n` by `n` matrix stored in `np` by `np` array) is input as the identity matrix. If the eigenvectors of a matrix that has been reduced by `tred2` are required, then `z` is input as the matrix output by `tred2`. In either case, the `k`th column of `z` returns the normalized eigenvector corresponding to `d(k)`.

```
      INTEGER i,iter,k,l,m
      REAL b,c,dd,f,g,p,r,s,pythag
```

```
      do 11 i=2,n                                  Convenient to renumber the elements of e.
        e(i-1)=e(i)
      enddo 11
      e(n)=0.
      do 15 l=1,n
        iter=0
1       do 12 m=l,n-1                              Look for a single small subdiagonal element
          dd=abs(d(m))+abs(d(m+1))                   to split the matrix.
          if (abs(e(m))+dd.eq.dd) goto 2
        enddo 12
        m=n
2       if(m.ne.l)then
          if(iter.eq.30)pause 'too many iterations in tqli'
          iter=iter+1
          g=(d(l+1)-d(l))/(2.*e(l))                Form shift.
          r=pythag(g,1.)
          g=d(m)-d(l)+e(l)/(g+sign(r,g))           This is d_m - k_s.
          s=1.
          c=1.
          p=0.
          do 14 i=m-1,l,-1                         A plane rotation as in the original QL, fol-
            f=s*e(i)                                 lowed by Givens rotations to restore tridi-
            b=c*e(i)                                 agonal form.
            r=pythag(f,g)
            e(i+1)=r
            if(r.eq.0.)then                        Recover from underflow.
              d(i+1)=d(i+1)-p
              e(m)=0.
              goto 1
            endif
            s=f/r
            c=g/r
            g=d(i+1)-p
            r=(d(i)-g)*s+2.*c*b
            p=s*r
            d(i+1)=g+p
            g=c*r-b
C   Omit lines from here ...
            do 13 k=1,n                            Form eigenvectors.
              f=z(k,i+1)
              z(k,i+1)=s*z(k,i)+c*f
              z(k,i)=c*z(k,i)-s*f
            enddo 13
C   ... to here when finding only eigenvalues.
          enddo 14
          d(l)=d(l)-p
          e(l)=g
          e(m)=0.
          goto 1
        endif
      enddo 15
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 331–335. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.6.6. [4]

11.4 Hermitian Matrices

The complex analog of a real, symmetric matrix is a Hermitian matrix, satisfying equation (11.0.4). Jacobi transformations can be used to find eigenvalues and eigenvectors, as also can Householder reduction to tridiagonal form followed by QL iteration. Complex versions of the previous routines `jacobi`, `tred2`, and `tqli` are quite analogous to their real counterparts. For working routines, consult [1,2].

An alternative, using the routines in this book, is to convert the Hermitian problem to a real, symmetric one: If $\mathbf{C} = \mathbf{A} + i\mathbf{B}$ is a Hermitian matrix, then the $n \times n$ complex eigenvalue problem

$$(\mathbf{A} + i\mathbf{B}) \cdot (\mathbf{u} + i\mathbf{v}) = \lambda(\mathbf{u} + i\mathbf{v}) \tag{11.4.1}$$

is equivalent to the $2n \times 2n$ real problem

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \tag{11.4.2}$$

Note that the $2n \times 2n$ matrix in (11.4.2) is symmetric: $\mathbf{A}^T = \mathbf{A}$ and $\mathbf{B}^T = -\mathbf{B}$ if $\mathbf{C}$ is Hermitian.

Corresponding to a given eigenvalue λ, the vector

$$\begin{bmatrix} -\mathbf{v} \\ \mathbf{u} \end{bmatrix} \tag{11.4.3}$$

is also an eigenvector, as you can verify by writing out the two matrix equations implied by (11.4.2). Thus if $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of $\mathbf{C}$, then the $2n$ eigenvalues of the augmented problem (11.4.2) are $\lambda_1, \lambda_1, \lambda_2, \lambda_2, \ldots, \lambda_n, \lambda_n$; each, in other words, is repeated twice. The eigenvectors are pairs of the form $\mathbf{u} + i\mathbf{v}$ and $i(\mathbf{u} + i\mathbf{v})$; that is, they are the same up to an inessential phase. Thus we solve the augmented problem (11.4.2), and choose one eigenvalue and eigenvector from each pair. These give the eigenvalues and eigenvectors of the original matrix $\mathbf{C}$.

Working with the augmented matrix requires a factor of 2 more storage than the original complex matrix. In principle, a complex algorithm is also a factor of 2 more efficient in computer time than is the solution of the augmented problem. In practice, most complex implementations do not achieve this factor unless they are written entirely in real arithmetic. (Good library routines always do this.)

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.6.6. [4]

11.4 Hermitian Matrices

The complex analog of a real, symmetric matrix is a Hermitian matrix, satisfying equation (11.0.4). Jacobi transformations can be used to find eigenvalues and eigenvectors, as also can Householder reduction to tridiagonal form followed by QL iteration. Complex versions of the previous routines `jacobi`, `tred2`, and `tqli` are quite analogous to their real counterparts. For working routines, consult [1,2].

An alternative, using the routines in this book, is to convert the Hermitian problem to a real, symmetric one: If $\mathbf{C} = \mathbf{A} + i\mathbf{B}$ is a Hermitian matrix, then the $n \times n$ complex eigenvalue problem

$$(\mathbf{A} + i\mathbf{B}) \cdot (\mathbf{u} + i\mathbf{v}) = \lambda(\mathbf{u} + i\mathbf{v}) \tag{11.4.1}$$

is equivalent to the $2n \times 2n$ real problem

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \tag{11.4.2}$$

Note that the $2n \times 2n$ matrix in (11.4.2) is symmetric: $\mathbf{A}^T = \mathbf{A}$ and $\mathbf{B}^T = -\mathbf{B}$ if $\mathbf{C}$ is Hermitian.

Corresponding to a given eigenvalue λ, the vector

$$\begin{bmatrix} -\mathbf{v} \\ \mathbf{u} \end{bmatrix} \tag{11.4.3}$$

is also an eigenvector, as you can verify by writing out the two matrix equations implied by (11.4.2). Thus if $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of $\mathbf{C}$, then the $2n$ eigenvalues of the augmented problem (11.4.2) are $\lambda_1, \lambda_1, \lambda_2, \lambda_2, \ldots, \lambda_n, \lambda_n$; each, in other words, is repeated twice. The eigenvectors are pairs of the form $\mathbf{u} + i\mathbf{v}$ and $i(\mathbf{u} + i\mathbf{v})$; that is, they are the same up to an inessential phase. Thus we solve the augmented problem (11.4.2), and choose one eigenvalue and eigenvector from each pair. These give the eigenvalues and eigenvectors of the original matrix $\mathbf{C}$.

Working with the augmented matrix requires a factor of 2 more storage than the original complex matrix. In principle, a complex algorithm is also a factor of 2 more efficient in computer time than is the solution of the augmented problem. In practice, most complex implementations do not achieve this factor unless they are written entirely in real arithmetic. (Good library routines always do this.)

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

11.5 Reduction of a General Matrix to Hessenberg Form

The algorithms for symmetric matrices, given in the preceding sections, are highly satisfactory in practice. By contrast, it is impossible to design equally satisfactory algorithms for the nonsymmetric case. There are two reasons for this. First, the eigenvalues of a nonsymmetric matrix can be very sensitive to small changes in the matrix elements. Second, the matrix itself can be defective, so that there is no complete set of eigenvectors. We emphasize that these difficulties are intrinsic properties of certain nonsymmetric matrices, and no numerical procedure can "cure" them. The best we can hope for are procedures that don't exacerbate such problems.

The presence of rounding error can only make the situation worse. With finite-precision arithmetic, one cannot even design a foolproof algorithm to determine whether a given matrix is defective or not. Thus current algorithms generally *try* to find a *complete* set of eigenvectors, and rely on the user to inspect the results. If any eigenvectors are almost parallel, the matrix is probably defective.

Apart from referring you to the literature, and to the collected routines in [1,2], we are going to sidestep the problem of eigenvectors, giving algorithms for eigenvalues only. If you require just a few eigenvectors, you can read §11.7 and consider finding them by inverse iteration. We consider the problem of finding *all* eigenvectors of a nonsymmetric matrix as lying beyond the scope of this book.

Balancing

The sensitivity of eigenvalues to rounding errors during the execution of some algorithms can be reduced by the procedure of *balancing*. The errors in the eigensystem found by a numerical procedure are generally proportional to the Euclidean norm of the matrix, that is, to the square root of the sum of the squares of the elements. The idea of balancing is to use similarity transformations to make corresponding rows and columns of the matrix have comparable norms, thus reducing the overall norm of the matrix while leaving the eigenvalues unchanged. A symmetric matrix is already balanced.

Balancing is a procedure with of order N^2 operations. Thus, the time taken by the procedure `balanc`, given below, should never be more than a few percent of the total time required to find the eigenvalues. It is therefore recommended that you *always* balance nonsymmetric matrices. It never hurts, and it can substantially improve the accuracy of the eigenvalues computed for a badly balanced matrix.

The actual algorithm used is due to Osborne, as discussed in [1]. It consists of a sequence of similarity transformations by diagonal matrices **D**. To avoid introducing rounding errors during the balancing process, the elements of **D** are restricted to be exact powers of the radix base employed for floating-point arithmetic (i.e., 2 for most machines, but 16 for IBM mainframe architectures). The output is a matrix that is balanced in the norm given by summing the absolute magnitudes of the matrix elements. This is more efficient than using the Euclidean norm, and equally effective: A large reduction in one norm implies a large reduction in the other.

Note that if the off-diagonal elements of any row or column of a matrix are all zero, then the diagonal element is an eigenvalue. If the eigenvalue happens to

11.5 Reduction of a General Matrix to Hessenberg Form

The algorithms for symmetric matrices, given in the preceding sections, are highly satisfactory in practice. By contrast, it is impossible to design equally satisfactory algorithms for the nonsymmetric case. There are two reasons for this. First, the eigenvalues of a nonsymmetric matrix can be very sensitive to small changes in the matrix elements. Second, the matrix itself can be defective, so that there is no complete set of eigenvectors. We emphasize that these difficulties are intrinsic properties of certain nonsymmetric matrices, and no numerical procedure can "cure" them. The best we can hope for are procedures that don't exacerbate such problems.

The presence of rounding error can only make the situation worse. With finite-precision arithmetic, one cannot even design a foolproof algorithm to determine whether a given matrix is defective or not. Thus current algorithms generally *try* to find a *complete* set of eigenvectors, and rely on the user to inspect the results. If any eigenvectors are almost parallel, the matrix is probably defective.

Apart from referring you to the literature, and to the collected routines in [1,2], we are going to sidestep the problem of eigenvectors, giving algorithms for eigenvalues only. If you require just a few eigenvectors, you can read §11.7 and consider finding them by inverse iteration. We consider the problem of finding *all* eigenvectors of a nonsymmetric matrix as lying beyond the scope of this book.

Balancing

The sensitivity of eigenvalues to rounding errors during the execution of some algorithms can be reduced by the procedure of *balancing*. The errors in the eigensystem found by a numerical procedure are generally proportional to the Euclidean norm of the matrix, that is, to the square root of the sum of the squares of the elements. The idea of balancing is to use similarity transformations to make corresponding rows and columns of the matrix have comparable norms, thus reducing the overall norm of the matrix while leaving the eigenvalues unchanged. A symmetric matrix is already balanced.

Balancing is a procedure with of order N^2 operations. Thus, the time taken by the procedure `balanc`, given below, should never be more than a few percent of the total time required to find the eigenvalues. It is therefore recommended that you *always* balance nonsymmetric matrices. It never hurts, and it can substantially improve the accuracy of the eigenvalues computed for a badly balanced matrix.

The actual algorithm used is due to Osborne, as discussed in [1]. It consists of a sequence of similarity transformations by diagonal matrices **D**. To avoid introducing rounding errors during the balancing process, the elements of **D** are restricted to be exact powers of the radix base employed for floating-point arithmetic (i.e., 2 for most machines, but 16 for IBM mainframe architectures). The output is a matrix that is balanced in the norm given by summing the absolute magnitudes of the matrix elements. This is more efficient than using the Euclidean norm, and equally effective: A large reduction in one norm implies a large reduction in the other.

Note that if the off-diagonal elements of any row or column of a matrix are all zero, then the diagonal element is an eigenvalue. If the eigenvalue happens to

be ill-conditioned (sensitive to small changes in the matrix elements), it will have relatively large errors when determined by the routine hqr (§11.6). Had we merely inspected the matrix beforehand, we could have determined the isolated eigenvalue exactly and then deleted the corresponding row and column from the matrix. You should consider whether such a pre-inspection might be useful in your application. (For symmetric matrices, the routines we gave will determine isolated eigenvalues accurately in all cases.)

The routine balanc does not keep track of the accumulated similarity transformation of the original matrix, since we will only be concerned with finding eigenvalues of nonsymmetric matrices, not eigenvectors. Consult [1-3] if you want to keep track of the transformation.

```
      SUBROUTINE balanc(a,n,np)
      INTEGER n,np
      REAL a(np,np),RADIX,SQRDX
      PARAMETER (RADIX=2.,SQRDX=RADIX**2)
        Given an n by n matrix a stored in an array of physical dimensions np by np, this routine
        replaces it by a balanced matrix with identical eigenvalues. A symmetric matrix is already
        balanced and is unaffected by this procedure. The parameter RADIX should be the machine's
        floating-point radix.
      INTEGER i,j,last
      REAL c,f,g,r,s
1     continue
        last=1
        do 14 i=1,n                    Calculate row and column norms.
          c=0.
          r=0.
          do 11 j=1,n
            if(j.ne.i)then
              c=c+abs(a(j,i))
              r=r+abs(a(i,j))
            endif
          enddo 11
          if(c.ne.0..and.r.ne.0.)then        If both are nonzero,
            g=r/RADIX
            f=1.
            s=c+r
2           if(c.lt.g)then                   find the integer power of the machine radix that
              f=f*RADIX                      comes closest to balancing the matrix.
              c=c*SQRDX
            goto 2
            endif
            g=r*RADIX
3           if(c.gt.g)then
              f=f/RADIX
              c=c/SQRDX
            goto 3
            endif
            if((c+r)/f.lt.0.95*s)then
              last=0
              g=1./f
              do 12 j=1,n                  Apply similarity transformation.
                a(i,j)=a(i,j)*g
              enddo 12
              do 13 j=1,n
                a(j,i)=a(j,i)*f
              enddo 13
            endif
          endif
        enddo 14
```

```
if(last.eq.0)goto 1
return
END
```

Reduction to Hessenberg Form

The strategy for finding the eigensystem of a general matrix parallels that of the symmetric case. First we reduce the matrix to a simpler form, and then we perform an iterative procedure on the simplified matrix. The simpler structure we use here is called *Hessenberg* form. An *upper Hessenberg* matrix has zeros everywhere below the diagonal except for the first subdiagonal row. For example, in the 6×6 case, the nonzero elements are:

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \end{bmatrix}$$

By now you should be able to tell at a glance that such a structure can be achieved by a sequence of Householder transformations, each one zeroing the required elements in a column of the matrix. Householder reduction to Hessenberg form is in fact an accepted technique. An alternative, however, is a procedure analogous to Gaussian elimination with pivoting. We will use this elimination procedure since it is about a factor of 2 more efficient than the Householder method, and also since we want to teach you the method. It is possible to construct matrices for which the Householder reduction, being orthogonal, is stable and elimination is not, but such matrices are extremely rare in practice.

Straight Gaussian elimination is not a similarity transformation of the matrix. Accordingly, the actual elimination procedure used is slightly different. Before the rth stage, the original matrix $\mathbf{A} \equiv \mathbf{A}_1$ has become $\mathbf{A}_r$, which is upper Hessenberg in its first $r-1$ rows and columns. The rth stage then consists of the following sequence of operations:

- Find the element of maximum magnitude in the rth column below the diagonal. If it is zero, skip the next two "bullets" and the stage is done. Otherwise, suppose the maximum element was in row r'.
- Interchange rows r' and $r+1$. This is the pivoting procedure. To make the permutation a similarity transformation, also interchange columns r' and $r+1$.
- For $i = r+2, r+3, \ldots, N$, compute the multiplier

$$n_{i,r+1} \equiv \frac{a_{ir}}{a_{r+1,r}}$$

  Subtract $n_{i,r+1}$ times row $r+1$ from row i. To make the elimination a similarity transformation, also *add* $n_{i,r+1}$ times column i to column $r+1$.

A total of $N-2$ such stages are required.

When the magnitudes of the matrix elements vary over many orders, you should try to rearrange the matrix so that the largest elements are in the top left-hand corner. This reduces the roundoff error, since the reduction proceeds from left to right.

Since we are concerned only with eigenvalues, the routine `elmhes` does not keep track of the accumulated similarity transformation. The operation count is about $5N^3/6$ for large N.

```
SUBROUTINE elmhes(a,n,np)
INTEGER n,np
REAL a(np,np)
    Reduction to Hessenberg form by the elimination method. The real, nonsymmetric, n by
    n matrix a, stored in an array of physical dimensions np by np, is replaced by an upper
    Hessenberg matrix with identical eigenvalues. Recommended, but not required, is that this
    routine be preceded by balanc. On output, the Hessenberg matrix is in elements a(i,j)
    with i ≤ j+1. Elements with i > j+1 are to be thought of as zero, but are returned with
    random values.
INTEGER i,j,m
REAL x,y
do 17 m=2,n-1                   m is called r + 1 in the text.
    x=0.
    i=m
    do 11 j=m,n                 Find the pivot.
        if(abs(a(j,m-1)).gt.abs(x))then
            x=a(j,m-1)
            i=j
        endif
    enddo 11
    if(i.ne.m)then              Interchange rows and columns.
        do 12 j=m-1,n
            y=a(i,j)
            a(i,j)=a(m,j)
            a(m,j)=y
        enddo 12
        do 13 j=1,n
            y=a(j,i)
            a(j,i)=a(j,m)
            a(j,m)=y
        enddo 13
    endif
    if(x.ne.0.)then             Carry out the elimination.
        do 16 i=m+1,n
            y=a(i,m-1)
            if(y.ne.0.)then
                y=y/x
                a(i,m-1)=y
                do 14 j=m,n
                    a(i,j)=a(i,j)-y*a(m,j)
                enddo 14
                do 15 j=1,n
                    a(j,m)=a(j,m)+y*a(j,i)
                enddo 15
            endif
        enddo 16
    endif
enddo 17
return
END
```

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.5.4. [3]

11.6 The QR Algorithm for Real Hessenberg Matrices

Recall the following relations for the QR algorithm with shifts:

$$\mathbf{Q}_s \cdot (\mathbf{A}_s - k_s \mathbf{1}) = \mathbf{R}_s \tag{11.6.1}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper triangular, and

$$\begin{aligned} \mathbf{A}_{s+1} &= \mathbf{R}_s \cdot \mathbf{Q}_s^T + k_s \mathbf{1} \\ &= \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \end{aligned} \tag{11.6.2}$$

The QR transformation preserves the upper Hessenberg form of the original matrix $\mathbf{A} \equiv \mathbf{A}_1$, and the workload on such a matrix is $O(n^2)$ per iteration as opposed to $O(n^3)$ on a general matrix. As $s \to \infty$, $\mathbf{A}_s$ converges to a form where the eigenvalues are either isolated on the diagonal or are eigenvalues of a 2×2 submatrix on the diagonal.

As we pointed out in §11.3, shifting is essential for rapid convergence. A key difference here is that a nonsymmetric real matrix can have complex eigenvalues. This means that good choices for the shifts k_s may be complex, apparently necessitating complex arithmetic.

Complex arithmetic can be avoided, however, by a clever trick. The trick depends on a result analogous to the lemma we used for implicit shifts in §11.3. The lemma we need here states that if $\mathbf{B}$ is a nonsingular matrix such that

$$\mathbf{B} \cdot \mathbf{Q} = \mathbf{Q} \cdot \mathbf{H} \tag{11.6.3}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{H}$ is upper Hessenberg, then $\mathbf{Q}$ and $\mathbf{H}$ are fully determined by the first column of $\mathbf{Q}$. (The determination is unique if $\mathbf{H}$ has positive subdiagonal elements.) The lemma can be proved by induction analogously to the proof given for tridiagonal matrices in §11.3.

The lemma is used in practice by taking two steps of the QR algorithm, either with two real shifts k_s and k_{s+1}, or with complex conjugate values k_s and $k_{s+1} = k_s{}^*$. This gives a real matrix $\mathbf{A}_{s+2}$, where

$$\mathbf{A}_{s+2} = \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \cdot \mathbf{Q}_{s+1}^T. \tag{11.6.4}$$

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §6.5.4. [3]

11.6 The QR Algorithm for Real Hessenberg Matrices

Recall the following relations for the QR algorithm with shifts:

$$\mathbf{Q}_s \cdot (\mathbf{A}_s - k_s \mathbf{1}) = \mathbf{R}_s \tag{11.6.1}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper triangular, and

$$\begin{aligned} \mathbf{A}_{s+1} &= \mathbf{R}_s \cdot \mathbf{Q}_s^T + k_s \mathbf{1} \\ &= \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \end{aligned} \tag{11.6.2}$$

The QR transformation preserves the upper Hessenberg form of the original matrix $\mathbf{A} \equiv \mathbf{A}_1$, and the workload on such a matrix is $O(n^2)$ per iteration as opposed to $O(n^3)$ on a general matrix. As $s \to \infty$, $\mathbf{A}_s$ converges to a form where the eigenvalues are either isolated on the diagonal or are eigenvalues of a 2×2 submatrix on the diagonal.

As we pointed out in §11.3, shifting is essential for rapid convergence. A key difference here is that a nonsymmetric real matrix can have complex eigenvalues. This means that good choices for the shifts k_s may be complex, apparently necessitating complex arithmetic.

Complex arithmetic can be avoided, however, by a clever trick. The trick depends on a result analogous to the lemma we used for implicit shifts in §11.3. The lemma we need here states that if $\mathbf{B}$ is a nonsingular matrix such that

$$\mathbf{B} \cdot \mathbf{Q} = \mathbf{Q} \cdot \mathbf{H} \tag{11.6.3}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{H}$ is upper Hessenberg, then $\mathbf{Q}$ and $\mathbf{H}$ are fully determined by the first column of $\mathbf{Q}$. (The determination is unique if $\mathbf{H}$ has positive subdiagonal elements.) The lemma can be proved by induction analogously to the proof given for tridiagonal matrices in §11.3.

The lemma is used in practice by taking two steps of the QR algorithm, either with two real shifts k_s and k_{s+1}, or with complex conjugate values k_s and $k_{s+1} = k_s{}^*$. This gives a real matrix $\mathbf{A}_{s+2}$, where

$$\mathbf{A}_{s+2} = \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \cdot \mathbf{Q}_{s+1}^T. \tag{11.6.4}$$

The $\mathbf{Q}$'s are determined by

$$\mathbf{A}_s - k_s\mathbf{1} = \mathbf{Q}_s^T \cdot \mathbf{R}_s \tag{11.6.5}$$
$$\mathbf{A}_{s+1} = \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \tag{11.6.6}$$
$$\mathbf{A}_{s+1} - k_{s+1}\mathbf{1} = \mathbf{Q}_{s+1}^T \cdot \mathbf{R}_{s+1} \tag{11.6.7}$$

Using (11.6.6), equation (11.6.7) can be rewritten

$$\mathbf{A}_s - k_{s+1}\mathbf{1} = \mathbf{Q}_s^T \cdot \mathbf{Q}_{s+1}^T \cdot \mathbf{R}_{s+1} \cdot \mathbf{Q}_s \tag{11.6.8}$$

Hence, if we define

$$\mathbf{M} = (\mathbf{A}_s - k_{s+1}\mathbf{1}) \cdot (\mathbf{A}_s - k_s\mathbf{1}) \tag{11.6.9}$$

equations (11.6.5) and (11.6.8) give

$$\mathbf{R} = \mathbf{Q} \cdot \mathbf{M} \tag{11.6.10}$$

where

$$\mathbf{Q} = \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \tag{11.6.11}$$
$$\mathbf{R} = \mathbf{R}_{s+1} \cdot \mathbf{R}_s \tag{11.6.12}$$

Equation (11.6.4) can be rewritten

$$\mathbf{A}_s \cdot \mathbf{Q}^T = \mathbf{Q}^T \cdot \mathbf{A}_{s+2} \tag{11.6.13}$$

Thus suppose we can somehow find an upper Hessenberg matrix $\mathbf{H}$ such that

$$\mathbf{A}_s \cdot \overline{\mathbf{Q}}^T = \overline{\mathbf{Q}}^T \cdot \mathbf{H} \tag{11.6.14}$$

where $\overline{\mathbf{Q}}$ is orthogonal. If $\overline{\mathbf{Q}}^T$ has the same first column as $\mathbf{Q}^T$ (i.e., $\overline{\mathbf{Q}}$ has the same first row as $\mathbf{Q}$), then $\overline{\mathbf{Q}} = \mathbf{Q}$ and $\mathbf{A}_{s+2} = \mathbf{H}$.

The first row of $\mathbf{Q}$ is found as follows. Equation (11.6.10) shows that $\mathbf{Q}$ is the orthogonal matrix that triangularizes the real matrix $\mathbf{M}$. Any real matrix can be triangularized by premultiplying it by a sequence of Householder matrices $\mathbf{P}_1$ (acting on the first column), $\mathbf{P}_2$ (acting on the second column), $\ldots$, $\mathbf{P}_{n-1}$. Thus $\mathbf{Q} = \mathbf{P}_{n-1} \cdots \mathbf{P}_2 \cdot \mathbf{P}_1$, and the first row of $\mathbf{Q}$ is the first row of $\mathbf{P}_1$ since $\mathbf{P}_i$ is an $(i-1) \times (i-1)$ identity matrix in the top left-hand corner. We now must find $\overline{\mathbf{Q}}$ satisfying (11.6.14) whose first row is that of $\mathbf{P}_1$.

The Householder matrix $\mathbf{P}_1$ is determined by the first column of $\mathbf{M}$. Since $\mathbf{A}_s$ is upper Hessenberg, equation (11.6.9) shows that the first column of $\mathbf{M}$ has the form $[p_1, q_1, r_1, 0, ..., 0]^T$, where

$$\begin{aligned} p_1 &= a_{11}^2 - a_{11}(k_s + k_{s+1}) + k_s k_{s+1} + a_{12}a_{21} \\ q_1 &= a_{21}(a_{11} + a_{22} - k_s - k_{s+1}) \\ r_1 &= a_{21}a_{32} \end{aligned} \tag{11.6.15}$$

Hence

$$\mathbf{P}_1 = 1 - 2\mathbf{w}_1 \cdot \mathbf{w}_1^T \tag{11.6.16}$$

where $\mathbf{w}_1$ has only its first 3 elements nonzero (cf. equation 11.2.5). The matrix $\mathbf{P}_1 \cdot \mathbf{A}_s \cdot \mathbf{P}_1^T$ is therefore upper Hessenberg with 3 extra elements:

$$\mathbf{P}_1 \cdot \mathbf{A}_1 \cdot \mathbf{P}_1^T = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ \mathbf{x} & \times & \times & \times & \times & \times & \times \\ \mathbf{x} & \mathbf{x} & \times & \times & \times & \times & \times \\ & & & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix} \tag{11.6.17}$$

This matrix can be restored to upper Hessenberg form without affecting the first row by a sequence of Householder similarity transformations. The first such Householder matrix, $\mathbf{P}_2$, acts on elements 2, 3, and 4 in the first column, annihilating elements 3 and 4. This produces a matrix of the same form as (11.6.17), with the 3 extra elements appearing one column over:

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times & \times \\ & \mathbf{x} & \times & \times & \times & \times & \times \\ & \mathbf{x} & \mathbf{x} & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix} \tag{11.6.18}$$

Proceeding in this way up to $\mathbf{P}_{n-1}$, we see that at each stage the Householder matrix $\mathbf{P}_r$ has a vector $\mathbf{w}_r$ that is nonzero only in elements r, $r+1$, and $r+2$. These elements are determined by the elements r, $r+1$, and $r+2$ in the $(r-1)$st column of the current matrix. Note that the preliminary matrix $\mathbf{P}_1$ has the same structure as $\mathbf{P}_2, \ldots, \mathbf{P}_{n-1}$.

The result is that

$$\mathbf{P}_{n-1} \cdots \mathbf{P}_2 \cdot \mathbf{P}_1 \cdot \mathbf{A}_s \cdot \mathbf{P}_1^T \cdot \mathbf{P}_2^T \cdots \mathbf{P}_{n-1}^T = \mathbf{H} \tag{11.6.19}$$

where $\mathbf{H}$ is upper Hessenberg. Thus

$$\overline{\mathbf{Q}} = \mathbf{Q} = \mathbf{P}_{n-1} \cdots \mathbf{P}_2 \cdot \mathbf{P}_1 \tag{11.6.20}$$

and

$$\mathbf{A}_{s+2} = \mathbf{H} \tag{11.6.21}$$

The shifts of origin at each stage are taken to be the eigenvalues of the 2×2 matrix in the bottom right-hand corner of the current $\mathbf{A}_s$. This gives

$$\begin{aligned} k_s + k_{s+2} &= a_{n-1,n-1} + a_{nn} \\ k_s k_{s+1} &= a_{n-1,n-1} a_{nn} - a_{n-1,n} a_{n,n-1} \end{aligned} \tag{11.6.22}$$

Substituting (11.6.22) in (11.6.15), we get

$$\begin{aligned} p_1 &= a_{21}\{[(a_{nn}-a_{11})(a_{n-1,n-1}-a_{11})-a_{n-1,n}a_{n,n-1}]/a_{21}+a_{12}\} \\ q_1 &= a_{21}[a_{22}-a_{11}-(a_{nn}-a_{11})-(a_{n-1,n-1}-a_{11})] \\ r_1 &= a_{21}a_{32} \end{aligned} \tag{11.6.23}$$

We have judiciously grouped terms to reduce possible roundoff when there are small off-diagonal elements. Since only the ratios of elements are relevant for a Householder transformation, we can omit the factor a_{21} from (11.6.23).

In summary, to carry out a double QR step we construct the Householder matrices $\mathbf{P}_r,\ r = 1,\ldots,n-1$. For $\mathbf{P}_1$ we use p_1, q_1, and r_1 given by (11.6.23). For the remaining matrices, p_r, q_r, and r_r are determined by the $(r, r-1)$, $(r+1, r-1)$, and $(r+2, r-1)$ elements of the current matrix. The number of arithmetic operations can be reduced by writing the nonzero elements of the $2\mathbf{w}\cdot\mathbf{w}^T$ part of the Householder matrix in the form

$$2\mathbf{w}\cdot\mathbf{w}^T = \begin{bmatrix} (p\pm s)/(\pm s) \\ q/(\pm s) \\ r/(\pm s) \end{bmatrix} \cdot [\,1 \quad q/(p\pm s) \quad r/(p\pm s)\,] \tag{11.6.24}$$

where

$$s^2 = p^2+q^2+r^2 \tag{11.6.25}$$

(We have simply divided each element by a piece of the normalizing factor; cf. the equations in §11.2.)

If we proceed in this way, convergence is usually very fast. There are two possible ways of terminating the iteration for an eigenvalue. First, if $a_{n,n-1}$ becomes "negligible," then a_{nn} is an eigenvalue. We can then delete the nth row and column of the matrix and look for the next eigenvalue. Alternatively, $a_{n-1,n-2}$ may become negligible. In this case the eigenvalues of the 2×2 matrix in the lower right-hand corner may be taken to be eigenvalues. We delete the nth and $(n-1)$st rows and columns of the matrix and continue.

The test for convergence to an eigenvalue is combined with a test for negligible subdiagonal elements that allows splitting of the matrix into submatrices. We find the largest i such that $a_{i,i-1}$ is negligible. If $i = n$, we have found a single eigenvalue. If $i = n-1$, we have found two eigenvalues. Otherwise we continue the iteration on the submatrix in rows i to n (i being set to unity if there is no small subdiagonal element).

After determining i, the submatrix in rows i to n is examined to see if the *product* of any two consecutive subdiagonal elements is small enough that we can work with an even smaller submatrix, starting say in row m. We start with $m = n-2$ and decrement it down to $i+1$, computing p, q, and r according to equations (11.6.23) with 1 replaced by m and 2 by $m+1$. If these were indeed the elements of the special "first" Householder matrix in a double QR step, then applying the Householder matrix would lead to nonzero elements in positions $(m+1, m-1)$, $(m+2, m-1)$, and $(m+2, m)$. We require that the first two of these elements be

small compared with the local diagonal elements $a_{m-1,m-1}$, a_{mm} and $a_{m+1,m+1}$. A satisfactory approximate criterion is

$$|a_{m,m-1}|(|q|+|r|) \ll |p|(|a_{m+1,m+1}|+|a_{mm}|+|a_{m-1,m-1}|) \tag{11.6.26}$$

Very rarely, the procedure described so far will fail to converge. On such matrices, experience shows that if one double step is performed with any shifts that are of order the norm of the matrix, convergence is subsequently very rapid. Accordingly, if ten iterations occur without determining an eigenvalue, the usual shifts are replaced for the next iteration by shifts defined by

$$\begin{aligned} k_s + k_{s+1} &= 1.5 \times (|a_{n,n-1}| + |a_{n-1,n-2}|) \\ k_s k_{s+1} &= (|a_{n,n-1}| + |a_{n-1,n-2}|)^2 \end{aligned} \tag{11.6.27}$$

The factor 1.5 was arbitrarily chosen to lessen the likelihood of an "unfortunate" choice of shifts. This strategy is repeated after 20 unsuccessful iterations. After 30 unsuccessful iterations, the routine reports failure.

The operation count for the QR algorithm described here is $\sim 5k^2$ per iteration, where k is the current size of the matrix. The typical average number of iterations per eigenvalue is ~ 1.8, so the total operation count for all the eigenvalues is $\sim 3n^3$. This estimate neglects any possible efficiency due to splitting or sparseness of the matrix.

The following routine `hqr` is based algorithmically on the above description, in turn following the implementations in [1,2].

```
      SUBROUTINE hqr(a,n,np,wr,wi)
      INTEGER n,np
      REAL a(np,np),wi(np),wr(np)
          Finds all eigenvalues of an n by n upper Hessenberg matrix a that is stored in an np by np
          array. On input a can be exactly as output from elmhes §11.5; on output it is destroyed.
          The real and imaginary parts of the eigenvalues are returned in wr and wi, respectively.
      INTEGER i,its,j,k,l,m,nn
      REAL anorm,p,q,r,s,t,u,v,w,x,y,z
      anorm=0.                                  Compute matrix norm for possible use
      do 12 i=1,n                                   in locating single small subdiagonal
          do 11 j=max(i-1,1),n                      element.
              anorm=anorm+abs(a(i,j))
          enddo 11
      enddo 12
      nn=n
      t=0.                                      Gets changed only by an exceptional shift.
1     if(nn.ge.1)then                           Begin search for next eigenvalue.
          its=0
2         do 13 l=nn,2,-1                       Begin iteration: look for single small sub-
              s=abs(a(l-1,l-1))+abs(a(l,l))         diagonal element.
              if(s.eq.0.)s=anorm
              if(abs(a(l,l-1))+s.eq.s)goto 3
          enddo 13
          l=1
3         x=a(nn,nn)
          if(l.eq.nn)then                       One root found.
              wr(nn)=x+t
              wi(nn)=0.
              nn=nn-1
          else
              y=a(nn-1,nn-1)
              w=a(nn,nn-1)*a(nn-1,nn)
```

```
      if(l.eq.nn-1)then                    Two roots found...
        p=0.5*(y-x)
        q=p**2+w
        z=sqrt(abs(q))
        x=x+t
        if(q.ge.0.)then                    ...a real pair.
          z=p+sign(z,p)
          wr(nn)=x+z
          wr(nn-1)=wr(nn)
          if(z.ne.0.)wr(nn)=x-w/z
          wi(nn)=0.
          wi(nn-1)=0.
        else                               ...a complex pair.
          wr(nn)=x+p
          wr(nn-1)=wr(nn)
          wi(nn)=z
          wi(nn-1)=-z
        endif
        nn=nn-2
      else                                 No roots found. Continue iteration.
        if(its.eq.30)pause 'too many iterations in hqr'
        if(its.eq.10.or.its.eq.20)then     Form exceptional shift.
          t=t+x
          do 14 i=1,nn
            a(i,i)=a(i,i)-x
          enddo 14
          s=abs(a(nn,nn-1))+abs(a(nn-1,nn-2))
          x=0.75*s
          y=x
          w=-0.4375*s**2
        endif
        its=its+1
        do 15 m=nn-2,l,-1                  Form shift and then look for 2 consecu-
          z=a(m,m)                           tive small subdiagonal elements.
          r=x-z
          s=y-z
          p=(r*s-w)/a(m+1,m)+a(m,m+1)      Equation (11.6.23).
          q=a(m+1,m+1)-z-r-s
          r=a(m+2,m+1)
          s=abs(p)+abs(q)+abs(r)           Scale to prevent overflow or underflow.
          p=p/s
          q=q/s
          r=r/s
          if(m.eq.l)goto 4
          u=abs(a(m,m-1))*(abs(q)+abs(r))
          v=abs(p)*(abs(a(m-1,m-1))+abs(z)+abs(a(m+1,m+1)))
          if(u+v.eq.v)goto 4               Equation (11.6.26).
        enddo 15
4       do 16 i=m+2,nn
          a(i,i-2)=0.
          if (i.ne.m+2) a(i,i-3)=0.
        enddo 16
        do 19 k=m,nn-1                     Double QR step on rows l to nn and
          if(k.ne.m)then                     columns m to nn.
            p=a(k,k-1)                     Begin setup of Householder vector.
            q=a(k+1,k-1)
            r=0.
            if(k.ne.nn-1)r=a(k+2,k-1)
            x=abs(p)+abs(q)+abs(r)
            if(x.ne.0.)then
              p=p/x                        Scale to prevent overflow or underflow.
              q=q/x
              r=r/x
            endif
```

```
                endif
                s=sign(sqrt(p**2+q**2+r**2),p)
                if(s.ne.0.)then
                    if(k.eq.m)then
                        if(l.ne.m)a(k,k-1)=-a(k,k-1)
                    else
                        a(k,k-1)=-s*x
                    endif
                    p=p+s                          Equations (11.6.24).
                    x=p/s
                    y=q/s
                    z=r/s
                    q=q/p
                    r=r/p
                    do 17 j=k,nn                   Row modification.
                        p=a(k,j)+q*a(k+1,j)
                        if(k.ne.nn-1)then
                            p=p+r*a(k+2,j)
                            a(k+2,j)=a(k+2,j)-p*z
                        endif
                        a(k+1,j)=a(k+1,j)-p*y
                        a(k,j)=a(k,j)-p*x
                    enddo 17
                    do 18 i=l,min(nn,k+3)          Column modification.
                        p=x*a(i,k)+y*a(i,k+1)
                        if(k.ne.nn-1)then
                            p=p+z*a(i,k+2)
                            a(i,k+2)=a(i,k+2)-p*r
                        endif
                        a(i,k+1)=a(i,k+1)-p*q
                        a(i,k)=a(i,k)-p
                    enddo 18
                endif
            enddo 19
            goto 2                                 ...for next iteration on current eigenvalue.
        endif
    endif
goto 1                                             ...for next eigenvalue.
endif
return
END
```

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §7.5.

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

11.7 Improving Eigenvalues and/or Finding Eigenvectors by Inverse Iteration

The basic idea behind inverse iteration is quite simple. Let $\mathbf{y}$ be the solution of the linear system

$$(\mathbf{A} - \tau\mathbf{1}) \cdot \mathbf{y} = \mathbf{b} \tag{11.7.1}$$

where $\mathbf{b}$ is a random vector and τ is close to some eigenvalue λ of $\mathbf{A}$. Then the solution $\mathbf{y}$ will be close to the eigenvector corresponding to λ. The procedure can be iterated: Replace $\mathbf{b}$ by $\mathbf{y}$ and solve for a new $\mathbf{y}$, which will be even closer to the true eigenvector.

We can see why this works by expanding both $\mathbf{y}$ and $\mathbf{b}$ as linear combinations of the eigenvectors $\mathbf{x}_j$ of $\mathbf{A}$:

$$\mathbf{y} = \sum_j \alpha_j \mathbf{x}_j \qquad \mathbf{b} = \sum_j \beta_j \mathbf{x}_j \tag{11.7.2}$$

Then (11.7.1) gives

$$\sum_j \alpha_j(\lambda_j - \tau)\mathbf{x}_j = \sum_j \beta_j \mathbf{x}_j \tag{11.7.3}$$

so that

$$\alpha_j = \frac{\beta_j}{\lambda_j - \tau} \tag{11.7.4}$$

and

$$\mathbf{y} = \sum_j \frac{\beta_j \mathbf{x}_j}{\lambda_j - \tau} \tag{11.7.5}$$

If τ is close to λ_n, say, then provided β_n is not accidentally too small, $\mathbf{y}$ will be approximately $\mathbf{x}_n$, up to a normalization. Moreover, each iteration of this procedure gives another power of $\lambda_j - \tau$ in the denominator of (11.7.5). Thus the convergence is rapid for well-separated eigenvalues.

Suppose at the kth stage of iteration we are solving the equation

$$(\mathbf{A} - \tau_k\mathbf{1}) \cdot \mathbf{y} = \mathbf{b}_k \tag{11.7.6}$$

where $\mathbf{b}_k$ and τ_k are our current guesses for some eigenvector and eigenvalue of interest (let's say, $\mathbf{x}_n$ and λ_n). Normalize $\mathbf{b}_k$ so that $\mathbf{b}_k \cdot \mathbf{b}_k = 1$. The exact eigenvector and eigenvalue satisfy

$$\mathbf{A} \cdot \mathbf{x}_n = \lambda_n \mathbf{x}_n \tag{11.7.7}$$

so

$$(\mathbf{A} - \tau_k\mathbf{1}) \cdot \mathbf{x}_n = (\lambda_n - \tau_k)\mathbf{x}_n \tag{11.7.8}$$

Since $\mathbf{y}$ of (11.7.6) is an improved approximation to $\mathbf{x}_n$, we normalize it and set

$$\mathbf{b}_{k+1} = \frac{\mathbf{y}}{|\mathbf{y}|} \tag{11.7.9}$$

11.7 Improving Eigenvalues and/or Finding Eigenvectors by Inverse Iteration

The basic idea behind inverse iteration is quite simple. Let $\mathbf{y}$ be the solution of the linear system

$$(\mathbf{A} - \tau\mathbf{1}) \cdot \mathbf{y} = \mathbf{b} \tag{11.7.1}$$

where $\mathbf{b}$ is a random vector and τ is close to some eigenvalue λ of $\mathbf{A}$. Then the solution $\mathbf{y}$ will be close to the eigenvector corresponding to λ. The procedure can be iterated: Replace $\mathbf{b}$ by $\mathbf{y}$ and solve for a new $\mathbf{y}$, which will be even closer to the true eigenvector.

We can see why this works by expanding both $\mathbf{y}$ and $\mathbf{b}$ as linear combinations of the eigenvectors $\mathbf{x}_j$ of $\mathbf{A}$:

$$\mathbf{y} = \sum_j \alpha_j \mathbf{x}_j \qquad \mathbf{b} = \sum_j \beta_j \mathbf{x}_j \tag{11.7.2}$$

Then (11.7.1) gives

$$\sum_j \alpha_j(\lambda_j - \tau)\mathbf{x}_j = \sum_j \beta_j \mathbf{x}_j \tag{11.7.3}$$

so that

$$\alpha_j = \frac{\beta_j}{\lambda_j - \tau} \tag{11.7.4}$$

and

$$\mathbf{y} = \sum_j \frac{\beta_j \mathbf{x}_j}{\lambda_j - \tau} \tag{11.7.5}$$

If τ is close to λ_n, say, then provided β_n is not accidentally too small, $\mathbf{y}$ will be approximately $\mathbf{x}_n$, up to a normalization. Moreover, each iteration of this procedure gives another power of $\lambda_j - \tau$ in the denominator of (11.7.5). Thus the convergence is rapid for well-separated eigenvalues.

Suppose at the kth stage of iteration we are solving the equation

$$(\mathbf{A} - \tau_k\mathbf{1}) \cdot \mathbf{y} = \mathbf{b}_k \tag{11.7.6}$$

where $\mathbf{b}_k$ and τ_k are our current guesses for some eigenvector and eigenvalue of interest (let's say, $\mathbf{x}_n$ and λ_n). Normalize $\mathbf{b}_k$ so that $\mathbf{b}_k \cdot \mathbf{b}_k = 1$. The exact eigenvector and eigenvalue satisfy

$$\mathbf{A} \cdot \mathbf{x}_n = \lambda_n \mathbf{x}_n \tag{11.7.7}$$

so

$$(\mathbf{A} - \tau_k\mathbf{1}) \cdot \mathbf{x}_n = (\lambda_n - \tau_k)\mathbf{x}_n \tag{11.7.8}$$

Since $\mathbf{y}$ of (11.7.6) is an improved approximation to $\mathbf{x}_n$, we normalize it and set

$$\mathbf{b}_{k+1} = \frac{\mathbf{y}}{|\mathbf{y}|} \tag{11.7.9}$$

We get an improved estimate of the eigenvalue by substituting our improved guess $\mathbf{y}$ for $\mathbf{x}_n$ in (11.7.8). By (11.7.6), the left-hand side is $\mathbf{b}_k$, so calling λ_n our new value τ_{k+1}, we find

$$\tau_{k+1} = \tau_k + \frac{1}{\mathbf{b}_k \cdot \mathbf{y}} \tag{11.7.10}$$

While the above formulas look simple enough, in practice the implementation can be quite tricky. The first question to be resolved is *when* to use inverse iteration. Most of the computational load occurs in solving the linear system (11.7.6). Thus a possible strategy is first to reduce the matrix $\mathbf{A}$ to a special form that allows easy solution of (11.7.6). Tridiagonal form for symmetric matrices or Hessenberg for nonsymmetric are the obvious choices. Then apply inverse iteration to generate all the eigenvectors. While this is an $O(N^3)$ method for symmetric matrices, it is many times less efficient than the QL method given earlier. In fact, even the best inverse iteration packages are less efficient than the QL method as soon as more than about 25 percent of the eigenvectors are required. Accordingly, inverse iteration is generally used when one already has good eigenvalues and wants only a few selected eigenvectors.

You can write a simple inverse iteration routine yourself using LU decomposition to solve (11.7.6). You can decide whether to use the general LU algorithm we gave in Chapter 2 or whether to take advantage of tridiagonal or Hessenberg form. Note that, since the linear system (11.7.6) is nearly singular, you must be careful to use a version of LU decomposition like that in §2.3 which replaces a zero pivot with a very small number.

We have chosen not to give a general inverse iteration routine in this book, because it is quite cumbersome to take account of all the cases that can arise. Routines are given, for example, in [1,2]. If you use these, or write your own routine, you may appreciate the following pointers.

One starts by supplying an initial value τ_0 for the eigenvalue λ_n of interest. Choose a random normalized vector $\mathbf{b}_0$ as the initial guess for the eigenvector $\mathbf{x}_n$, and solve (11.7.6). The new vector $\mathbf{y}$ is bigger than $\mathbf{b}_0$ by a "growth factor" $|\mathbf{y}|$, which ideally should be large. Equivalently, the change in the eigenvalue, which by (11.7.10) is essentially $1/|\mathbf{y}|$, should be small. The following cases can arise:

- If the growth factor is too small initially, then we assume we have made a "bad" choice of random vector. This can happen not just because of a small β_n in (11.7.5), but also in the case of a defective matrix, when (11.7.5) does not even apply (see, e.g., [1] or [3] for details). We go back to the beginning and choose a new initial vector.
- The change $|\mathbf{b}_1 - \mathbf{b}_0|$ might be less than some tolerance ϵ. We can use this as a criterion for stopping, iterating until it is satisfied, with a maximum of 5 – 10 iterations, say.
- After a few iterations, if $|\mathbf{b}_{k+1} - \mathbf{b}_k|$ is not decreasing rapidly enough, we can try updating the eigenvalue according to (11.7.10). If $\tau_{k+1} = \tau_k$ to machine accuracy, we are not going to improve the eigenvector much more and can quit. Otherwise start another cycle of iterations with the new eigenvalue.

The reason we do not update the eigenvalue at every step is that when we solve the linear system (11.7.6) by LU decomposition, we can save the decomposition

if τ_k is fixed. We only need do the backsubstitution step each time we update $\mathbf{b}_k$. The number of iterations we decide to do with a fixed τ_k is a trade-off between the quadratic convergence but $O(N^3)$ workload for updating τ_k at each step and the linear convergence but $O(N^2)$ load for keeping τ_k fixed. If you have determined the eigenvalue by one of the routines given earlier in the chapter, it is probably correct to machine accuracy anyway, and you can omit updating it.

There are two different pathologies that can arise during inverse iteration. The first is multiple or closely spaced roots. This is more often a problem with symmetric matrices. Inverse iteration will find only one eigenvector for a given initial guess τ_0. A good strategy is to perturb the last few significant digits in τ_0 and then repeat the iteration. Usually this provides an independent eigenvector. Special steps generally have to be taken to ensure orthogonality of the linearly independent eigenvectors, whereas the Jacobi and QL algorithms automatically yield orthogonal eigenvectors even in the case of multiple eigenvalues.

The second problem, peculiar to nonsymmetric matrices, is the defective case. Unless one makes a "good" initial guess, the growth factor is small. Moreover, iteration does not improve matters. In this case, the remedy is to choose random initial vectors, solve (11.7.6) once, and quit as soon as *any* vector gives an acceptably large growth factor. Typically only a few trials are necessary.

One further complication in the nonsymmetric case is that a real matrix can have complex-conjugate pairs of eigenvalues. You will then have to use complex arithmetic to solve (11.7.6) for the complex eigenvectors. For any moderate-sized (or larger) nonsymmetric matrix, our recommendation is to avoid inverse iteration in favor of a QR method that includes the eigenvector computation in complex arithmetic. You will find routines for this in [1,2] and other places.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), p. 418. [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), p. 356. [3]

Chapter 12. Fast Fourier Transform

12.0 Introduction

A very large class of important computational problems falls under the general rubric of "Fourier transform methods" or "spectral methods." For some of these problems, the Fourier transform is simply an efficient computational tool for accomplishing certain common manipulations of data. In other cases, we have problems for which the Fourier transform (or the related "power spectrum") is itself of intrinsic interest. These two kinds of problems share a common methodology.

Largely for historical reasons the literature on Fourier and spectral methods has been disjoint from the literature on "classical" numerical analysis. Nowadays there is no justification for such a split. Fourier methods are commonplace in research and we shall not treat them as specialized or arcane. At the same time, we realize that many computer users have had relatively less experience with this field than with, say, differential equations or numerical integration. Therefore our summary of analytical results will be more complete. Numerical algorithms, per se, begin in §12.2. Various applications of Fourier transform methods are discussed in Chapter 13.

A physical process can be described either in the *time domain*, by the values of some quantity h as a function of time t, e.g., $h(t)$, or else in the *frequency domain*, where the process is specified by giving its amplitude H (generally a complex number indicating phase also) as a function of frequency f, that is $H(f)$, with $-\infty < f < \infty$. For many purposes it is useful to think of $h(t)$ and $H(f)$ as being two different *representations* of the *same* function. One goes back and forth between these two representations by means of the *Fourier transform* equations,

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt \\ h(t) &= \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} df \end{aligned} \tag{12.0.1}$$

If t is measured in seconds, then f in equation (12.0.1) is in cycles per second, or Hertz (the unit of frequency). However, the equations work with other units too. If h is a function of position x (in meters), H will be a function of inverse wavelength (cycles per meter), and so on. If you are trained as a physicist or mathematician, you are probably more used to using *angular frequency* ω, which is given in *radians* per sec. The relation between ω and f, $H(\omega)$ and $H(f)$ is

$$\omega \equiv 2\pi f \qquad H(\omega) \equiv [H(f)]_{f=\omega/2\pi} \tag{12.0.2}$$

and equation (12.0.1) looks like this

$$H(\omega) = \int_{-\infty}^{\infty} h(t) e^{i\omega t} dt$$
$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{-i\omega t} d\omega \tag{12.0.3}$$

We were raised on the ω-convention, but we changed! There are fewer factors of 2π to remember if you use the f-convention, especially when we get to discretely sampled data in §12.1.

From equation (12.0.1) it is evident at once that Fourier transformation is a *linear* operation. The transform of the sum of two functions is equal to the sum of the transforms. The transform of a constant times a function is that same constant times the transform of the function.

In the time domain, function $h(t)$ may happen to have one or more special symmetries It might be *purely real* or *purely imaginary* or it might be *even*, $h(t) = h(-t)$, or *odd*, $h(t) = -h(-t)$. In the frequency domain, these symmetries lead to relationships between $H(f)$ and $H(-f)$. The following table gives the correspondence between symmetries in the two domains:

If . . .	then . . .
$h(t)$ is real	$H(-f) = [H(f)]^*$
$h(t)$ is imaginary	$H(-f) = -[H(f)]^*$
$h(t)$ is even	$H(-f) = H(f)$ [i.e., $H(f)$ is even]
$h(t)$ is odd	$H(-f) = -H(f)$ [i.e., $H(f)$ is odd]
$h(t)$ is real and even	$H(f)$ is real and even
$h(t)$ is real and odd	$H(f)$ is imaginary and odd
$h(t)$ is imaginary and even	$H(f)$ is imaginary and even
$h(t)$ is imaginary and odd	$H(f)$ is real and odd

In subsequent sections we shall see how to use these symmetries to increase computational efficiency.

Here are some other elementary properties of the Fourier transform. (We'll use the "$\Longleftrightarrow$" symbol to indicate transform pairs.) If

$$h(t) \Longleftrightarrow H(f)$$

is such a pair, then other transform pairs are

$$h(at) \Longleftrightarrow \frac{1}{|a|} H\left(\frac{f}{a}\right) \qquad \text{"time scaling"} \tag{12.0.4}$$
$$\frac{1}{|b|} h\left(\frac{t}{b}\right) \Longleftrightarrow H(bf) \qquad \text{"frequency scaling"} \tag{12.0.5}$$
$$h(t - t_0) \Longleftrightarrow H(f)\, e^{2\pi i f t_0} \qquad \text{"time shifting"} \tag{12.0.6}$$
$$h(t)\, e^{-2\pi i f_0 t} \Longleftrightarrow H(f - f_0) \qquad \text{"frequency shifting"} \tag{12.0.7}$$

With two functions $h(t)$ and $g(t)$, and their corresponding Fourier transforms $H(f)$ and $G(f)$, we can form two combinations of special interest. The *convolution* of the two functions, denoted $g * h$, is defined by

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau)h(t-\tau)\, d\tau \tag{12.0.8}$$

Note that $g * h$ is a function in the time domain and that $g * h = h * g$. It turns out that the function $g * h$ is one member of a simple transform pair

$$g * h \Longleftrightarrow G(f)H(f) \qquad \text{"Convolution Theorem"} \tag{12.0.9}$$

In other words, the Fourier transform of the convolution is just the product of the individual Fourier transforms.

The *correlation* of two functions, denoted Corr(g, h), is defined by

$$\text{Corr}(g,h) \equiv \int_{-\infty}^{\infty} g(\tau+t)h(\tau)\, d\tau \tag{12.0.10}$$

The correlation is a function of t, which is called the *lag*. It therefore lies in the time domain, and it turns out to be one member of the transform pair:

$$\text{Corr}(g,h) \Longleftrightarrow G(f)H^*(f) \qquad \text{"Correlation Theorem"} \tag{12.0.11}$$

[More generally, the second member of the pair is $G(f)H(-f)$, but we are restricting ourselves to the usual case in which g and h are real functions, so we take the liberty of setting $H(-f) = H^*(f)$.] This result shows that multiplying the Fourier transform of one function by the complex conjugate of the Fourier transform of the other gives the Fourier transform of their correlation. The correlation of a function with itself is called its *autocorrelation*. In this case (12.0.11) becomes the transform pair

$$\text{Corr}(g,g) \Longleftrightarrow |G(f)|^2 \qquad \text{"Wiener-Khinchin Theorem"} \tag{12.0.12}$$

The *total power* in a signal is the same whether we compute it in the time domain or in the frequency domain. This result is known as *Parseval's theorem*:

$$\text{Total Power} \equiv \int_{-\infty}^{\infty} |h(t)|^2\, dt = \int_{-\infty}^{\infty} |H(f)|^2\, df \tag{12.0.13}$$

Frequently one wants to know "how much power" is contained in the frequency interval between f and $f + df$. In such circumstances one does not usually distinguish between positive and negative f, but rather regards f as varying from 0 ("zero frequency" or D.C.) to $+\infty$. In such cases, one defines the *one-sided power spectral density (PSD)* of the function h as

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \qquad 0 \le f < \infty \tag{12.0.14}$$

so that the total power is just the integral of $P_h(f)$ from $f = 0$ to $f = \infty$. When the function $h(t)$ is real, then the two terms in (12.0.14) are equal, so $P_h(f) = 2\,|H(f)|^2$.

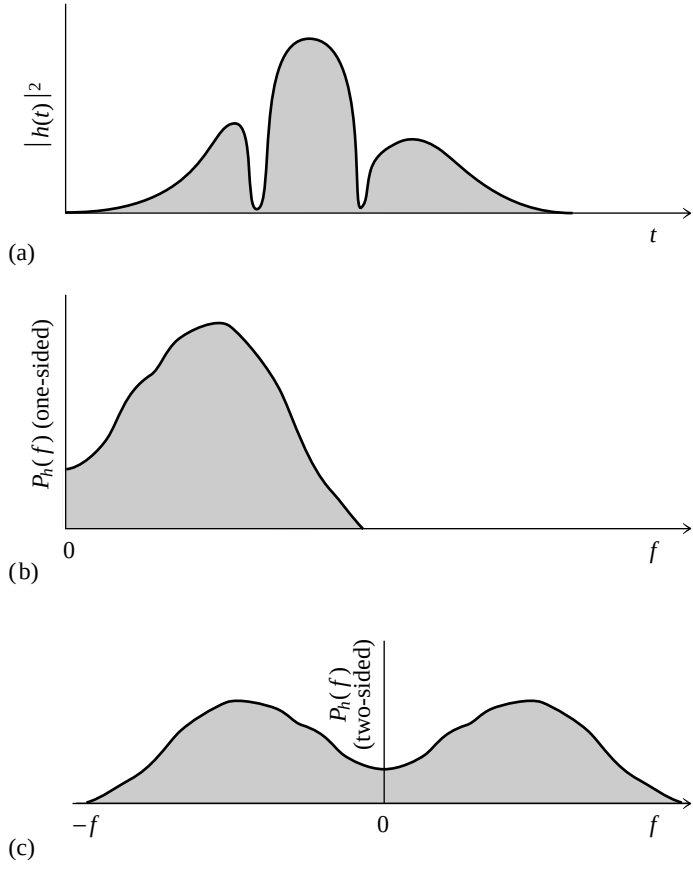


Figure 12.0.1. Normalizations of one- and two-sided power spectra. The area under the square of the function, (a), equals the area under its one-sided power spectrum at positive frequencies, (b), and also equals the area under its two-sided power spectrum at positive and negative frequencies, (c).

Be warned that one occasionally sees PSDs defined without this factor two. These, strictly speaking, are called *two-sided power spectral densities*, but some books are not careful about stating whether one- or two-sided is to be assumed. We will always use the one-sided density given by equation (12.0.14). Figure 12.0.1 contrasts the two conventions.

If the function $h(t)$ goes endlessly from $-\infty < t < \infty$, then its total power and power spectral density will, in general, be infinite. Of interest then is the *(one- or two-sided) power spectral density per unit time.* This is computed by taking a long, but finite, stretch of the function $h(t)$, computing its PSD [that is, the PSD of a function that equals $h(t)$ in the finite stretch but is zero everywhere else], and then dividing the resulting PSD by the length of the stretch used. Parseval's theorem in this case states that the integral of the one-sided PSD-per-unit-time over positive frequency is equal to the mean square amplitude of the signal $h(t)$.

You might well worry about how the PSD-per-unit-time, which is a function of frequency f, converges as one evaluates it using longer and longer stretches of data. This interesting question is the content of the subject of "power spectrum estimation," and will be considered below in §13.4–§13.7. A crude answer for

now is: The PSD-per-unit-time converges to finite values at all frequencies *except* those where $h(t)$ has a discrete sine-wave (or cosine-wave) component of finite amplitude. At those frequencies, it becomes a delta-function, i.e., a sharp spike, whose width gets narrower and narrower, but whose area converges to be the mean square amplitude of the discrete sine or cosine component at that frequency.

We have by now stated all of the analytical formalism that we will need in this chapter with one exception: In computational work, especially with experimental data, we are almost never given a continuous function $h(t)$ to work with, but are given, rather, a list of measurements of $h(t_i)$ for a discrete set of t_i's. The profound implications of this seemingly unimportant fact are the subject of the next section.

CITED REFERENCES AND FURTHER READING:

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.1 Fourier Transform of Discretely Sampled Data

In the most common situations, function $h(t)$ is sampled (i.e., its value is recorded) at evenly spaced intervals in time. Let Δ denote the time interval between consecutive samples, so that the sequence of sampled values is

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{12.1.1}$$

The reciprocal of the time interval Δ is called the *sampling rate*; if Δ is measured in seconds, for example, then the sampling rate is the number of samples recorded per second.

Sampling Theorem and Aliasing

For any sampling interval Δ, there is also a special frequency f_c, called the *Nyquist critical frequency*, given by

$$f_c \equiv \frac{1}{2\Delta} \tag{12.1.2}$$

If a sine wave of the Nyquist critical frequency is sampled at its positive peak value, then the next sample will be at its negative trough value, the sample after that at the positive peak again, and so on. Expressed otherwise: *Critical sampling of a sine wave is two sample points per cycle.* One frequently chooses to measure time in units of the sampling interval Δ. In this case the Nyquist critical frequency is just the constant 1/2.

The Nyquist critical frequency is important for two related, but distinct, reasons. One is good news, and the other bad news. First the good news. It is the remarkable

now is: The PSD-per-unit-time converges to finite values at all frequencies *except* those where $h(t)$ has a discrete sine-wave (or cosine-wave) component of finite amplitude. At those frequencies, it becomes a delta-function, i.e., a sharp spike, whose width gets narrower and narrower, but whose area converges to be the mean square amplitude of the discrete sine or cosine component at that frequency.

We have by now stated all of the analytical formalism that we will need in this chapter with one exception: In computational work, especially with experimental data, we are almost never given a continuous function $h(t)$ to work with, but are given, rather, a list of measurements of $h(t_i)$ for a discrete set of t_i's. The profound implications of this seemingly unimportant fact are the subject of the next section.

CITED REFERENCES AND FURTHER READING:

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.1 Fourier Transform of Discretely Sampled Data

In the most common situations, function $h(t)$ is sampled (i.e., its value is recorded) at evenly spaced intervals in time. Let Δ denote the time interval between consecutive samples, so that the sequence of sampled values is

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{12.1.1}$$

The reciprocal of the time interval Δ is called the *sampling rate*; if Δ is measured in seconds, for example, then the sampling rate is the number of samples recorded per second.

Sampling Theorem and Aliasing

For any sampling interval Δ, there is also a special frequency f_c, called the *Nyquist critical frequency*, given by

$$f_c \equiv \frac{1}{2\Delta} \tag{12.1.2}$$

If a sine wave of the Nyquist critical frequency is sampled at its positive peak value, then the next sample will be at its negative trough value, the sample after that at the positive peak again, and so on. Expressed otherwise: *Critical sampling of a sine wave is two sample points per cycle.* One frequently chooses to measure time in units of the sampling interval Δ. In this case the Nyquist critical frequency is just the constant 1/2.

The Nyquist critical frequency is important for two related, but distinct, reasons. One is good news, and the other bad news. First the good news. It is the remarkable

fact known as the *sampling theorem*: If a continuous function $h(t)$, sampled at an interval Δ, happens to be *bandwidth limited* to frequencies smaller in magnitude than f_c, i.e., if $H(f) = 0$ for all $|f| \geq f_c$, then the function $h(t)$ is *completely determined* by its samples h_n. In fact, $h(t)$ is given explicitly by the formula

$$h(t) = \Delta \sum_{n=-\infty}^{+\infty} h_n \frac{\sin[2\pi f_c(t - n\Delta)]}{\pi(t - n\Delta)} \tag{12.1.3}$$

This is a remarkable theorem for many reasons, among them that it shows that the "information content" of a bandwidth limited function is, in some sense, infinitely smaller than that of a general continuous function. Fairly often, one is dealing with a signal that is known on physical grounds to be bandwidth limited (or at least approximately bandwidth limited). For example, the signal may have passed through an amplifier with a known, finite frequency response. In this case, the sampling theorem tells us that the entire information content of the signal can be recorded by sampling it at a rate Δ^{-1} equal to twice the maximum frequency passed by the amplifier (cf. 12.1.2).

Now the bad news. The bad news concerns the effect of sampling a continuous function that is *not* bandwidth limited to less than the Nyquist critical frequency. In that case, it turns out that all of the power spectral density that lies outside of the frequency range $-f_c < f < f_c$ is spuriously moved into that range. This phenomenon is called *aliasing*. Any frequency component outside of the frequency range $(-f_c, f_c)$ is *aliased* (falsely translated) into that range by the very act of discrete sampling. You can readily convince yourself that two waves $\exp(2\pi i f_1 t)$ and $\exp(2\pi i f_2 t)$ give the same samples at an interval Δ if and only if f_1 and f_2 differ by a multiple of $1/\Delta$, which is just the width in frequency of the range $(-f_c, f_c)$. There is little that you can do to remove aliased power once you have discretely sampled a signal. The way to overcome aliasing is to (i) know the natural bandwidth limit of the signal — or else enforce a known limit by analog filtering of the continuous signal, and then (ii) sample at a rate sufficiently rapid to give at least two points per cycle of the highest frequency present. Figure 12.1.1 illustrates these considerations.

To put the best face on this, we can take the alternative point of view: If a continuous function has been competently sampled, then, when we come to estimate its Fourier transform from the discrete samples, we can *assume* (or rather we *might as well* assume) that its Fourier transform is equal to zero outside of the frequency range in between $-f_c$ and f_c. Then we look to the Fourier transform to tell whether the continuous function *has* been competently sampled (aliasing effects minimized). We do this by looking to see whether the Fourier transform is already approaching zero as the frequency approaches f_c from below, or $-f_c$ from above. If, on the contrary, the transform is going towards some finite value, then chances are that components outside of the range have been folded back over onto the critical range.

Discrete Fourier Transform

We now estimate the Fourier transform of a function from a finite number of its sampled points. Suppose that we have N consecutive sampled values

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1 \tag{12.1.4}$$

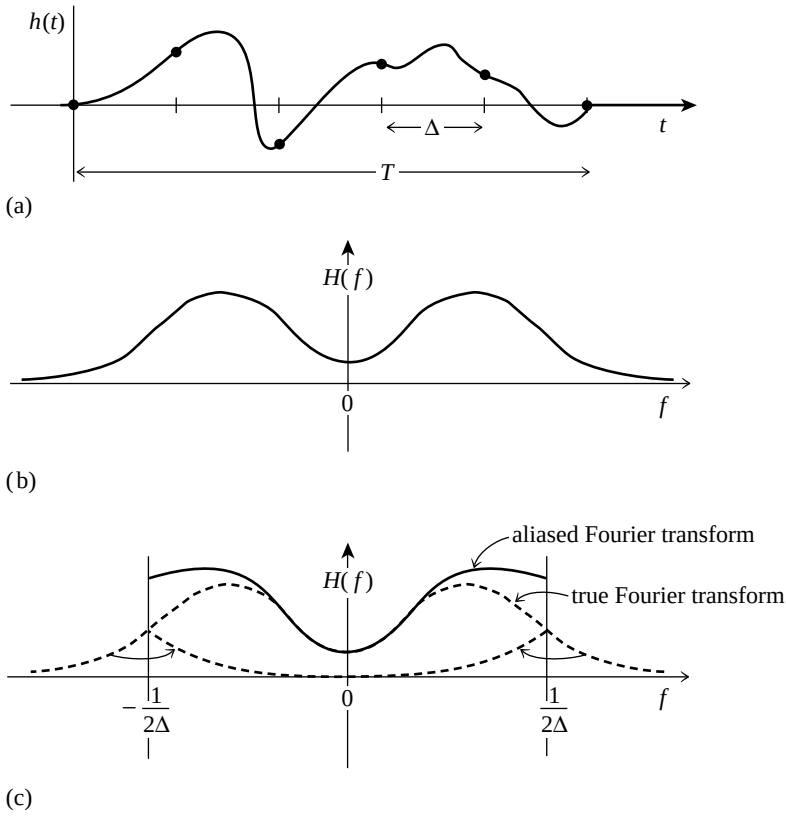


Figure 12.1.1. The continuous function shown in (a) is nonzero only for a finite interval of time T. It follows that its Fourier transform, whose modulus is shown schematically in (b), is not bandwidth limited but has finite amplitude for all frequencies. If the original function is sampled with a sampling interval Δ, as in (a), then the Fourier transform (c) is defined only between plus and minus the Nyquist critical frequency. Power outside that range is folded over or "aliased" into the range. The effect can be eliminated only by low-pass filtering the original function *before sampling*.

so that the sampling interval is Δ. To make things simpler, let us also suppose that N is even. If the function $h(t)$ is nonzero only in a finite interval of time, then that whole interval of time is supposed to be contained in the range of the N points given. Alternatively, if the function $h(t)$ goes on forever, then the sampled points are supposed to be at least "typical" of what $h(t)$ looks like at all other times.

With N numbers of input, we will evidently be able to produce no more than N independent numbers of output. So, instead of trying to estimate the Fourier transform $H(f)$ at all values of f in the range $-f_c$ to f_c, let us seek estimates only at the discrete values

$$f_n \equiv \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, \ldots, \frac{N}{2} \tag{12.1.5}$$

The extreme values of n in (12.1.5) correspond exactly to the lower and upper limits of the Nyquist critical frequency range. If you are really on the ball, you will have noticed that there are $N+1$, not N, values of n in (12.1.5); it will turn out that the two extreme values of n are not independent (in fact they are equal), but all the others are. This reduces the count to N.

The remaining step is to approximate the integral in (12.0.1) by a discrete sum:

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k \, e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n / N} \tag{12.1.6}$$

Here equations (12.1.4) and (12.1.5) have been used in the final equality. The final summation in equation (12.1.6) is called the *discrete Fourier transform* of the N points h_k. Let us denote it by H_n,

$$H_n \equiv \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n / N} \tag{12.1.7}$$

The discrete Fourier transform maps N complex numbers (the h_k's) into N complex numbers (the H_n's). It does not depend on any dimensional parameter, such as the time scale Δ. The relation (12.1.6) between the discrete Fourier transform of a set of numbers and their continuous Fourier transform when they are viewed as samples of a continuous function sampled at an interval Δ can be rewritten as

$$H(f_n) \approx \Delta H_n \tag{12.1.8}$$

where f_n is given by (12.1.5).

Up to now we have taken the view that the index n in (12.1.7) varies from $-N/2$ to $N/2$ (cf. 12.1.5). You can easily see, however, that (12.1.7) is periodic in n, with period N. Therefore, $H_{-n} = H_{N-n} \quad n = 1, 2, \ldots$. With this conversion in mind, one generally lets the n in H_n vary from 0 to $N-1$ (one complete period). Then n and k (in h_k) vary exactly over the same range, so the mapping of N numbers into N numbers is manifest. When this convention is followed, you must remember that zero frequency corresponds to $n = 0$, positive frequencies $0 < f < f_c$ correspond to values $1 \le n \le N/2 - 1$, while negative frequencies $-f_c < f < 0$ correspond to $N/2 + 1 \le n \le N - 1$. The value $n = N/2$ corresponds to *both* $f = f_c$ and $f = -f_c$.

The discrete Fourier transform has symmetry properties almost exactly the same as the continuous Fourier transform. For example, all the symmetries in the table following equation (12.0.3) hold if we read h_k for $h(t)$, H_n for $H(f)$, and H_{N-n} for $H(-f)$. (Likewise, "even" and "odd" in time refer to whether the values h_k at k and $N - k$ are identical or the negative of each other.)

The formula for the discrete *inverse* Fourier transform, which recovers the set of h_k's exactly from the H_n's is:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \, e^{-2\pi i k n / N} \tag{12.1.9}$$

Notice that the only differences between (12.1.9) and (12.1.7) are (i) changing the sign in the exponential, and (ii) dividing the answer by N. This means that a routine for calculating discrete Fourier transforms can also, with slight modification, calculate the inverse transforms.

The discrete form of Parseval's theorem is

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \tag{12.1.10}$$

There are also discrete analogs to the convolution and correlation theorems (equations 12.0.9 and 12.0.11), but we shall defer them to §13.1 and §13.2, respectively.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.2 Fast Fourier Transform (FFT)

How much computation is involved in computing the discrete Fourier transform (12.1.7) of N points? For many years, until the mid-1960s, the standard answer was this: Define W as the complex number

$$W \equiv e^{2\pi i/N} \tag{12.2.1}$$

Then (12.1.7) can be written as

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{12.2.2}$$

In other words, the vector of h_k's is multiplied by a matrix whose (n, k)th element is the constant W to the power $n \times k$. The matrix multiplication produces a vector result whose components are the H_n's. This matrix multiplication evidently requires N^2 complex multiplications, plus a smaller number of operations to generate the required powers of W. So, the discrete Fourier transform appears to be an $O(N^2)$ process. These appearances are deceiving! The discrete Fourier transform can, in fact, be computed in $O(N \log_2 N)$ operations with an algorithm called the *fast Fourier transform*, or *FFT*. The difference between $N \log_2 N$ and N^2 is immense. With $N = 10^6$, for example, it is the difference between, roughly, 30 seconds of CPU time and 2 weeks of CPU time on a microsecond cycle time computer. The existence of an FFT algorithm became generally known only in the mid-1960s, from the work of J.W. Cooley and J.W. Tukey. Retrospectively, we now know (see [1]) that efficient methods for computing the DFT had been independently discovered, and in some cases implemented, by as many as a dozen individuals, starting with Gauss in 1805!

One "rediscovery" of the FFT, that of Danielson and Lanczos in 1942, provides one of the clearest derivations of the algorithm. Danielson and Lanczos showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms, each of length $N/2$. One of the two is formed from the

The discrete form of Parseval's theorem is

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \tag{12.1.10}$$

There are also discrete analogs to the convolution and correlation theorems (equations 12.0.9 and 12.0.11), but we shall defer them to §13.1 and §13.2, respectively.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.2 Fast Fourier Transform (FFT)

How much computation is involved in computing the discrete Fourier transform (12.1.7) of N points? For many years, until the mid-1960s, the standard answer was this: Define W as the complex number

$$W \equiv e^{2\pi i/N} \tag{12.2.1}$$

Then (12.1.7) can be written as

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{12.2.2}$$

In other words, the vector of h_k's is multiplied by a matrix whose (n, k)th element is the constant W to the power $n \times k$. The matrix multiplication produces a vector result whose components are the H_n's. This matrix multiplication evidently requires N^2 complex multiplications, plus a smaller number of operations to generate the required powers of W. So, the discrete Fourier transform appears to be an $O(N^2)$ process. These appearances are deceiving! The discrete Fourier transform can, in fact, be computed in $O(N \log_2 N)$ operations with an algorithm called the *fast Fourier transform*, or *FFT*. The difference between $N \log_2 N$ and N^2 is immense. With $N = 10^6$, for example, it is the difference between, roughly, 30 seconds of CPU time and 2 weeks of CPU time on a microsecond cycle time computer. The existence of an FFT algorithm became generally known only in the mid-1960s, from the work of J.W. Cooley and J.W. Tukey. Retrospectively, we now know (see [1]) that efficient methods for computing the DFT had been independently discovered, and in some cases implemented, by as many as a dozen individuals, starting with Gauss in 1805!

One "rediscovery" of the FFT, that of Danielson and Lanczos in 1942, provides one of the clearest derivations of the algorithm. Danielson and Lanczos showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms, each of length $N/2$. One of the two is formed from the

even-numbered points of the original N, the other from the odd-numbered points. The proof is simply this:

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\ &= F_k^e + W^k\, F_k^o \end{aligned} \tag{12.2.3}$$

In the last line, W is the same complex constant as in (12.2.1), F_k^e denotes the kth component of the Fourier transform of length $N/2$ formed from the even components of the original f_j's, while F_k^o is the corresponding transform of length $N/2$ formed from the odd components. Notice also that k in the last line of (12.2.3) varies from 0 to N, not just to $N/2$. Nevertheless, the transforms F_k^e and F_k^o are periodic in k with length $N/2$. So each is repeated through two cycles to obtain F_k.

The wonderful thing about the *Danielson-Lanczos Lemma* is that it can be used recursively. Having reduced the problem of computing F_k to that of computing F_k^e and F_k^o, we can do the same reduction of F_k^e to the problem of computing the transform of *its* $N/4$ even-numbered input data and $N/4$ odd-numbered data. In other words, we can define F_k^{ee} and F_k^{eo} to be the discrete Fourier transforms of the points which are respectively even-even and even-odd on the successive subdivisions of the data.

Although there are ways of treating other cases, by far the easiest case is the one in which the original N is an integer power of 2. In fact, we categorically recommend that you *only* use FFTs with N a power of two. If the length of your data set is not a power of two, pad it with zeros up to the next power of two. (We will give more sophisticated suggestions in subsequent sections below.) With this restriction on N, it is evident that we can continue applying the Danielson-Lanczos Lemma until we have subdivided the data all the way down to transforms of length 1. What is the Fourier transform of length one? It is just the identity operation that copies its one input number into its one output slot! In other words, for every pattern of $\log_2 N$ e's and o's, there is a one-point transform that is just one of the input numbers f_n

$$F_k^{eoeeoeo\cdots oee} = f_n \qquad \text{for some } n \tag{12.2.4}$$

(Of course this one-point transform actually does not depend on k, since it is periodic in k with period 1.)

The next trick is to figure out which value of n corresponds to which pattern of e's and o's in equation (12.2.4). The answer is: Reverse the pattern of e's and o's, then let $e = 0$ and $o = 1$, and you will have, *in binary* the value of n. Do you see why it works? It is because the successive subdivisions of the data into even and odd are tests of successive low-order (least significant) bits of n. This idea of *bit reversal* can be exploited in a very clever way which, along with the Danielson-Lanczos

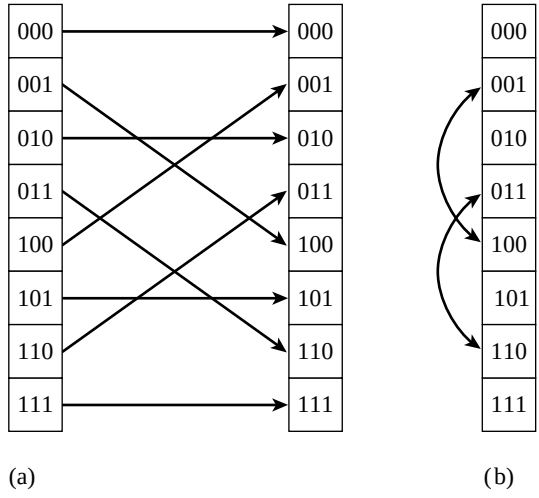


Figure 12.2.1. Reordering an array (here of length 8) by bit reversal, (a) between two arrays, versus (b) in place. Bit reversal reordering is a necessary part of the fast Fourier transform (FFT) algorithm.

Lemma, makes FFTs practical: Suppose we take the original vector of data f_j and rearrange it into bit-reversed order (see Figure 12.2.1), so that the individual numbers are in the order not of j, but of the number obtained by bit-reversing j. Then the bookkeeping on the recursive application of the Danielson-Lanczos Lemma becomes extraordinarily simple. The points as given are the one-point transforms. We combine adjacent pairs to get two-point transforms, then combine adjacent pairs of pairs to get 4-point transforms, and so on, until the first and second halves of the whole data set are combined into the final transform. Each combination takes of order N operations, and there are evidently $\log_2 N$ combinations, so the whole algorithm is of order $N \log_2 N$ (assuming, as is the case, that the process of sorting into bit-reversed order is no greater in order than $N \log_2 N$).

This, then, is the structure of an FFT algorithm: It has two sections. The first section sorts the data into bit-reversed order. Luckily this takes no additional storage, since it involves only swapping pairs of elements. (If k_1 is the bit reverse of k_2, then k_2 is the bit reverse of k_1.) The second section has an outer loop that is executed $\log_2 N$ times and calculates, in turn, transforms of length $2, 4, 8, \ldots, N$. For each stage of this process, two nested inner loops range over the subtransforms already computed and the elements of each transform, implementing the Danielson-Lanczos Lemma. The operation is made more efficient by restricting external calls for trigonometric sines and cosines to the outer loop, where they are made only $\log_2 N$ times. Computation of the sines and cosines of multiple angles is through simple recurrence relations in the inner loops (cf. 5.5.6).

The FFT routine given below is based on one originally written by N. M. Brenner. The input quantities are the number of complex data points (`nn`), the data array (`data`), and `isign`, which should be set to either ± 1 and is the sign of i in the exponential of equation (12.1.7). When `isign` is set to -1, the routine thus calculates the inverse transform (12.1.9) — except that it does not multiply by the normalizing factor $1/N$ that appears in that equation. You can do that yourself.

Notice that the argument `nn` is the number of *complex* data points, although

we avoid the use of complex arithmetic because of the inefficient implementations found on many computers. The actual length of the real array (data) is 2 times nn, with each complex value occupying two consecutive locations. In other words, data(1) is the real part of f_0, data(2) is the imaginary part of f_0, and so on up to data(2*nn-1), which is the real part of f_{N-1}, and data(2*nn), which is the imaginary part of f_{N-1}. The FFT routine returns the F_n's packed in exactly the same fashion, as nn complex numbers.

The real and imaginary parts of the zero frequency component F_0 are in data(1) and data(2); the smallest nonzero positive frequency has real and imaginary parts in data(3) and data(4); the smallest (in magnitude) nonzero negative frequency has real and imaginary parts in data(2*nn-1) and data(2*nn). Positive frequencies increasing in magnitude are stored in the real-imaginary pairs data(5), data(6) up to data(nn-1), data(nn). Negative frequencies of increasing magnitude are stored in data(2*nn-3), data(2*nn-2) down to data(nn+3), data(nn+4). Finally, the pair data(nn+1), data(nn+2) contain the real and imaginary parts of the one aliased point that contains the most positive and the most negative frequency. You should try to develop a familiarity with this storage arrangement of complex spectra, also shown in Figure 12.2.2, since it is the practical standard.

```
      SUBROUTINE four1(data,nn,isign)
      INTEGER isign,nn
      REAL data(2*nn)
          Replaces data(1:2*nn) by its discrete Fourier transform, if isign is input as 1; or replaces
          data(1:2*nn) by nn times its inverse discrete Fourier transform, if isign is input as -1.
          data is a complex array of length nn or, equivalently, a real array of length 2*nn. nn
          MUST be an integer power of 2 (this is not checked for!).
      INTEGER i,istep,j,m,mmax,n
      REAL tempi,tempr
      DOUBLE PRECISION theta,wi,wpi,wpr,wr,wtemp        Double precision for the trigonomet-
      n=2*nn                                              ric recurrences.
      j=1
      do 11 i=1,n,2                  This is the bit-reversal section of the routine.
          if(j.gt.i)then
              tempr=data(j)          Exchange the two complex numbers.
              tempi=data(j+1)
              data(j)=data(i)
              data(j+1)=data(i+1)
              data(i)=tempr
              data(i+1)=tempi
          endif
          m=n/2
1         if ((m.ge.2).and.(j.gt.m)) then
              j=j-m
              m=m/2
          goto 1
          endif
          j=j+m
      enddo 11
      mmax=2                         Here begins the Danielson-Lanczos section of the routine.
2     if (n.gt.mmax) then            Outer loop executed log2 nn times.
          istep=2*mmax
          theta=6.28318530717959d0/(isign*mmax)        Initialize for the trigonometric recur-
          wpr=-2.d0*sin(0.5d0*theta)**2                  rence.
          wpi=sin(theta)
          wr=1.d0
          wi=0.d0
          do 13 m=1,mmax,2           Here are the two nested inner loops.
```

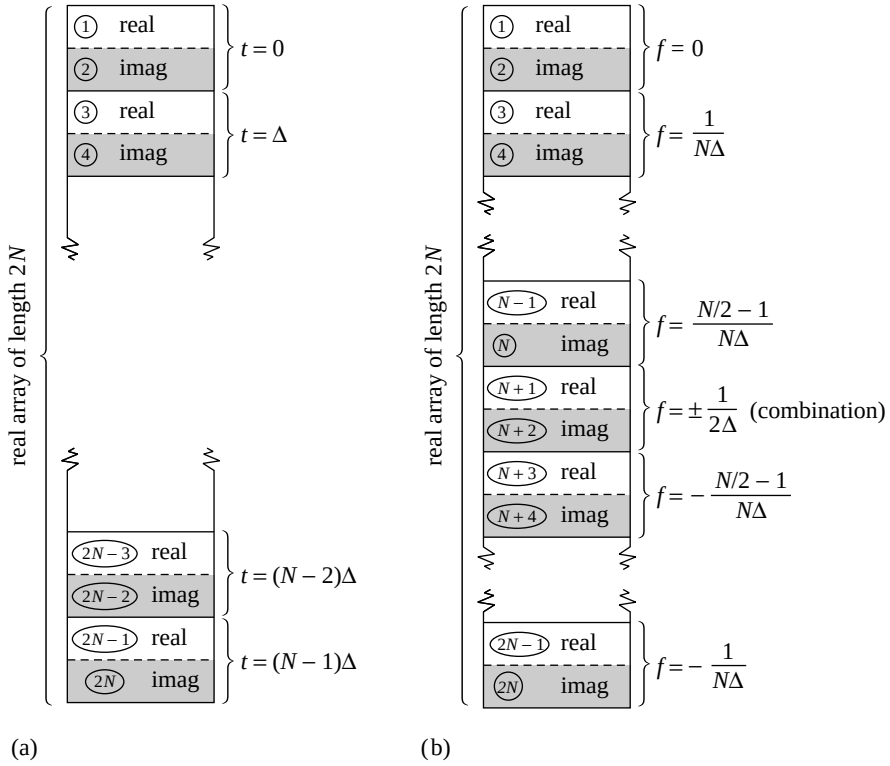


Figure 12.2.2. Input and output arrays for FFT. (a) The input array contains N (a power of 2) complex time samples in a real array of length $2N$, with real and imaginary parts alternating. (b) The output array contains the complex Fourier spectrum at N values of frequency. Real and imaginary parts again alternate. The array starts with zero frequency, works up to the most positive frequency (which is ambiguous with the most negative frequency). Negative frequencies follow, from the second-most negative up to the frequency just below zero.

```
        do 12 i=m,n,istep
            j=i+mmax            This is the Danielson-Lanczos formula:
            tempr=sngl(wr)*data(j)-sngl(wi)*data(j+1)
            tempi=sngl(wr)*data(j+1)+sngl(wi)*data(j)
            data(j)=data(i)-tempr
            data(j+1)=data(i+1)-tempi
            data(i)=data(i)+tempr
            data(i+1)=data(i+1)+tempi
        enddo 12
        wtemp=wr                Trigonometric recurrence.
        wr=wr*wpr-wi*wpi+wr
        wi=wi*wpr+wtemp*wpi+wi
    enddo 13
    mmax=istep
goto 2                          Not yet done.
endif                           All done.
return
END
```

(A double precision version of `four1`, named `dfour1`, is used by the routine `mpmul` in §20.6. You can easily make the conversion, or else get the converted routine from the *Numerical Recipes* diskette.)

Other FFT Algorithms

We should mention that there are a number of variants on the basic FFT algorithm given above. As we have seen, that algorithm first rearranges the input elements into bit-reverse order, then builds up the output transform in $\log_2 N$ iterations. In the literature, this sequence is called a *decimation-in-time* or *Cooley-Tukey* FFT algorithm. It is also possible to derive FFT algorithms that first go through a set of $\log_2 N$ iterations on the input data, and rearrange the *output* values into bit-reverse order. These are called *decimation-in-frequency* or *Sande-Tukey* FFT algorithms. For some applications, such as convolution (§13.1), one takes a data set into the Fourier domain and then, after some manipulation, back out again. In these cases it is possible to avoid all bit reversing. You use a decimation-in-frequency algorithm (without its bit reversing) to get into the "scrambled" Fourier domain, do your operations there, and then use an inverse algorithm (without *its* bit reversing) to get back to the time domain. While elegant in principle, this procedure does not in practice save much computation time, since the bit reversals represent only a small fraction of an FFT's operations count, and since most useful operations in the frequency domain require a knowledge of which points correspond to which frequencies.

Another class of FFTs subdivides the initial data set of length N not all the way down to the trivial transform of length 1, but rather only down to some other small power of 2, for example $N = 4$, *base-4 FFTs*, or $N = 8$, *base-8 FFTs*. These small transforms are then done by small sections of highly optimized coding which take advantage of special symmetries of that particular small N. For example, for $N = 4$, the trigonometric sines and cosines that enter are all ± 1 or 0, so many multiplications are eliminated, leaving largely additions and subtractions. These can be faster than simpler FFTs by some significant, but not overwhelming, factor, e.g., 20 or 30 percent.

There are also FFT algorithms for data sets of length N not a power of two. They work by using relations analogous to the Danielson-Lanczos Lemma to subdivide the initial problem into successively smaller problems, not by factors of 2, but by whatever small prime factors happen to divide N. The larger that the largest prime factor of N is, the worse this method works. If N is prime, then no subdivision is possible, and the user (whether he knows it or not) is taking a *slow* Fourier transform, of order N^2 instead of order $N \log_2 N$. Our advice is to stay clear of such FFT implementations, with perhaps one class of exceptions, the *Winograd Fourier transform algorithms*. Winograd algorithms are in some ways analogous to the base-4 and base-8 FFTs. Winograd has derived highly optimized codings for taking small-N discrete Fourier transforms, e.g., for $N = 2, 3, 4, 5, 7, 8, 11, 13, 16$. The algorithms also use a new and clever way of combining the subfactors. The method involves a reordering of the data both before the hierarchical processing and after it, but it allows a significant reduction in the number of multiplications in the algorithm. For some especially favorable values of N, the Winograd algorithms can be significantly (e.g., up to a factor of 2) faster than the simpler FFT algorithms of the nearest integer power of 2. This advantage in speed, however, must be weighed against the considerably more complicated data indexing involved in these transforms, and the fact that the Winograd transform cannot be done "in place."

Finally, an interesting class of transforms for doing convolutions quickly are number theoretic transforms. These schemes replace floating-point arithmetic with

integer arithmetic modulo some large prime N+1, and the Nth root of 1 by the modulo arithmetic equivalent. Strictly speaking, these are not *Fourier* transforms at all, but the properties are quite similar and computational speed can be far superior. On the other hand, their use is somewhat restricted to quantities like correlations and convolutions since the transform itself is not easily interpretable as a "frequency" spectrum.

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.).

Beauchamp, K.G. 1984, *Applications of Walsh Functions and Related Functions* (New York: Academic Press) [non-Fourier transforms].

Heideman, M.T., Johnson, D.H., and Burris, C.S. 1984, *IEEE ASSP Magazine*, pp. 14–21 (October).

12.3 FFT of Real Functions, Sine and Cosine Transforms

It happens frequently that the data whose FFT is desired consist of real-valued samples $f_j,\ j = 0 \ldots N-1$. To use `four1`, we put these into a complex array with all imaginary parts set to zero. The resulting transform $F_n,\ n = 0 \ldots N-1$ satisfies ${F_{N-n}}^* = F_n$. Since this complex-valued array has real values for F_0 and $F_{N/2}$, and $(N/2)-1$ other independent values $F_1 \ldots F_{N/2-1}$, it has the same $2(N/2-1)+2 = N$ "degrees of freedom" as the original, real data set. However, the use of the full complex FFT algorithm for real data is inefficient, both in execution time and in storage required. You would think that there is a better way.

There are *two* better ways. The first is "mass production": Pack two separate real functions into the input array in such a way that their individual transforms can be separated from the result. This is implemented in the program `twofft` below. This may remind you of a one-cent sale, at which you are coerced to purchase two of an item when you only need one. However, remember that for correlations and convolutions the Fourier transforms of two functions are involved, and this is a handy way to do them both at once. The second method is to pack the real input array cleverly, without extra zeros, into a complex array of half its length. One then performs a complex FFT on this shorter length; the trick is then to get the required answer out of the result. This is done in the program `realft` below.

integer arithmetic modulo some large prime N+1, and the Nth root of 1 by the modulo arithmetic equivalent. Strictly speaking, these are not *Fourier* transforms at all, but the properties are quite similar and computational speed can be far superior. On the other hand, their use is somewhat restricted to quantities like correlations and convolutions since the transform itself is not easily interpretable as a "frequency" spectrum.

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.).

Beauchamp, K.G. 1984, *Applications of Walsh Functions and Related Functions* (New York: Academic Press) [non-Fourier transforms].

Heideman, M.T., Johnson, D.H., and Burris, C.S. 1984, *IEEE ASSP Magazine*, pp. 14–21 (October).

12.3 FFT of Real Functions, Sine and Cosine Transforms

It happens frequently that the data whose FFT is desired consist of real-valued samples $f_j,\ j = 0 \ldots N-1$. To use `four1`, we put these into a complex array with all imaginary parts set to zero. The resulting transform $F_n,\ n = 0 \ldots N-1$ satisfies ${F_{N-n}}^* = F_n$. Since this complex-valued array has real values for F_0 and $F_{N/2}$, and $(N/2)-1$ other independent values $F_1 \ldots F_{N/2-1}$, it has the same $2(N/2-1)+2 = N$ "degrees of freedom" as the original, real data set. However, the use of the full complex FFT algorithm for real data is inefficient, both in execution time and in storage required. You would think that there is a better way.

There are *two* better ways. The first is "mass production": Pack two separate real functions into the input array in such a way that their individual transforms can be separated from the result. This is implemented in the program `twofft` below. This may remind you of a one-cent sale, at which you are coerced to purchase two of an item when you only need one. However, remember that for correlations and convolutions the Fourier transforms of two functions are involved, and this is a handy way to do them both at once. The second method is to pack the real input array cleverly, without extra zeros, into a complex array of half its length. One then performs a complex FFT on this shorter length; the trick is then to get the required answer out of the result. This is done in the program `realft` below.

Transform of Two Real Functions Simultaneously

First we show how to exploit the symmetry of the transform F_n to handle two real functions at once: Since the input data f_j are real, the components of the discrete Fourier transform satisfy

$$F_{N-n} = (F_n)^* \tag{12.3.1}$$

where the asterisk denotes complex conjugation. By the same token, the discrete Fourier transform of a purely imaginary set of g_j's has the opposite symmetry.

$$G_{N-n} = -(G_n)^* \tag{12.3.2}$$

Therefore we can take the discrete Fourier transform of two real functions each of length N simultaneously by packing the two data arrays as the real and imaginary parts, respectively, of the complex input array of `four1`. Then the resulting transform array can be unpacked into two complex arrays with the aid of the two symmetries. Routine `twofft` works out these ideas.

```
      SUBROUTINE twofft(data1,data2,fft1,fft2,n)
      INTEGER n
      REAL data1(n),data2(n)
      COMPLEX fft1(n),fft2(n)
C     USES four1
        Given two real input arrays data1(1:n) and data2(1:n), this routine calls four1 and
        returns two complex output arrays, fft1(1:n) and fft2(1:n), each of complex length n
        (i.e., real length 2*n), which contain the discrete Fourier transforms of the respective data
        arrays. n MUST be an integer power of 2.
      INTEGER j,n2
      COMPLEX h1,h2,c1,c2
      c1=cmplx(0.5,0.0)
      c2=cmplx(0.0,-0.5)
      do 11 j=1,n
          fft1(j)=cmplx(data1(j),data2(j))        Pack the two real arrays into one complex
      enddo 11                                        array.
      call four1(fft1,n,1)                        Transform the complex array.
      fft2(1)=cmplx(aimag(fft1(1)),0.0)
      fft1(1)=cmplx(real(fft1(1)),0.0)
      n2=n+2
      do 12 j=2,n/2+1
          h1=c1*(fft1(j)+conjg(fft1(n2-j)))       Use symmetries to separate the two trans-
          h2=c2*(fft1(j)-conjg(fft1(n2-j)))           forms.
          fft1(j)=h1                              Ship them out in two complex arrays.
          fft1(n2-j)=conjg(h1)
          fft2(j)=h2
          fft2(n2-j)=conjg(h2)
      enddo 12
      return
      END
```

What about the reverse process? Suppose you have two complex transform arrays, each of which has the symmetry (12.3.1), so that you know that the inverses of both transforms are real functions. Can you invert both in a single FFT? This is even easier than the other direction. Use the fact that the FFT is linear and form the sum of the first transform plus i times the second. Invert using `four1` with

isign $= -1$. The real and imaginary parts of the resulting complex array are the two desired real functions.

FFT of Single Real Function

To implement the second method, which allows us to perform the FFT of a *single* real function without redundancy, we split the data set in half, thereby forming two real arrays of half the size. We can apply the program above to these two, but of course the result will not be the transform of the original data. It will be a schizophrenic combination of two transforms, each of which has half of the information we need. Fortunately, this schizophrenia is treatable. It works like this:

The right way to split the original data is to take the even-numbered f_j as one data set, and the odd-numbered f_j as the other. The beauty of this is that we can take the original real array and treat it as a complex array h_j of half the length. The first data set is the real part of this array, and the second is the imaginary part, as prescribed for twofft. No repacking is required. In other words $h_j = f_{2j} + if_{2j+1}, \quad j = 0, \ldots, N/2 - 1$. We submit this to four1, and it will return a complex array $H_n = F_n^e + iF_n^o, \quad n = 0, \ldots, N/2 - 1$ with

$$\begin{aligned} F_n^e &= \sum_{k=0}^{N/2-1} f_{2k}\, e^{2\pi ikn/(N/2)} \\ F_n^o &= \sum_{k=0}^{N/2-1} f_{2k+1}\, e^{2\pi ikn/(N/2)} \end{aligned} \tag{12.3.3}$$

The discussion of program twofft tells you how to separate the two transforms F_n^e and F_n^o out of H_n. How do you work them into the transform F_n of the original data set f_j? Simply glance back at equation (12.2.3):

$$F_n = F_n^e + e^{2\pi in/N} F_n^o \qquad n = 0, \ldots, N-1 \tag{12.3.4}$$

Expressed directly in terms of the transform H_n of our real (masquerading as complex) data set, the result is

$$F_n = \frac{1}{2}(H_n + H_{N/2-n}{}^*) - \frac{i}{2}(H_n - H_{N/2-n}{}^*)e^{2\pi in/N} \qquad n = 0, \ldots, N-1 \tag{12.3.5}$$

A few remarks:

- Since ${F_{N-n}}^* = F_n$ there is no point in saving the entire spectrum. The positive frequency half is sufficient and can be stored in the same array as the original data. The operation can, in fact, be done in place.
- Even so, we need values $H_n,\ n = 0, \ldots, N/2$ whereas four1 returns only the values $n = 0, \ldots, N/2 - 1$. Symmetry to the rescue, $H_{N/2} = H_0$.
- The values F_0 and $F_{N/2}$ are real and independent. In order to actually get the entire F_n in the original array space, it is convenient to return $F_{N/2}$ as the imaginary part of F_0.

- Despite its complicated form, the process above is invertible. First peel $F_{N/2}$ out of F_0. Then construct

$$\begin{aligned} F_n^e &= \frac{1}{2}(F_n + F_{N/2-n}^*) \\ F_n^o &= \frac{1}{2}e^{-2\pi in/N}(F_n - F_{N/2-n}^*) \end{aligned} \qquad n = 0, \ldots, N/2-1 \tag{12.3.6}$$

  and use `four1` to find the inverse transform of $H_n = F_n^{(1)} + iF_n^{(2)}$. Surprisingly, the actual algebraic steps are virtually identical to those of the forward transform.

Here is a representation of what we have said:

```
      SUBROUTINE realft(data,n,isign)
      INTEGER isign,n
      REAL data(n)
C     USES four1
```

Calculates the Fourier transform of a set of `n` real-valued data points. Replaces this data (which is stored in array `data(1:n)`) by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements `data(1)` and `data(2)`, respectively. `n` must be a power of 2. This routine also calculates the inverse transform of a complex data array if it is the transform of real data. (Result in this case must be multiplied by `2/n`.)

```
      INTEGER i,i1,i2,i3,i4,n2p3
      REAL c1,c2,h1i,h1r,h2i,h2r,wis,wrs
      DOUBLE PRECISION theta,wi,wpi,wpr,
     *     wr,wtemp                          Double precision for the trigonometric recurrences.
      theta=3.141592653589793d0/dble(n/2)    Initialize the recurrence.
      c1=0.5
      if (isign.eq.1) then
        c2=-0.5
        call four1(data,n/2,+1)              The forward transform is here.
      else
        c2=0.5                               Otherwise set up for an inverse transform.
        theta=-theta
      endif
      wpr=-2.0d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      wr=1.0d0+wpr
      wi=wpi
      n2p3=n+3
      do 11 i=2,n/4                          Case i=1 done separately below.
        i1=2*i-1
        i2=i1+1
        i3=n2p3-i2
        i4=i3+1
        wrs=sngl(wr)
        wis=sngl(wi)
        h1r=c1*(data(i1)+data(i3))           The two separate transforms are separated out of
        h1i=c1*(data(i2)-data(i4))             data.
        h2r=-c2*(data(i2)+data(i4))
        h2i=c2*(data(i1)-data(i3))
        data(i1)=h1r+wrs*h2r-wis*h2i         Here they are recombined to form the true trans-
        data(i2)=h1i+wrs*h2i+wis*h2r           form of the original real data.
        data(i3)=h1r-wrs*h2r+wis*h2i
        data(i4)=-h1i+wrs*h2i+wis*h2r
        wtemp=wr                             The recurrence.
        wr=wr*wpr-wi*wpi+wr
        wi=wi*wpr+wtemp*wpi+wi
      enddo 11
```

```
if (isign.eq.1) then
    h1r=data(1)
    data(1)=h1r+data(2)
    data(2)=h1r-data(2)                Squeeze the first and last data together to get
else                                       them all within the original array.
    h1r=data(1)
    data(1)=c1*(h1r+data(2))
    data(2)=c1*(h1r-data(2))
    call four1(data,n/2,-1)            This is the inverse transform for the case isign=-1.
endif
return
END
```

Fast Sine and Cosine Transforms

Among their other uses, the Fourier transforms of functions can be used to solve differential equations (see §19.4). The most common boundary conditions for the solutions are 1) they have the value zero at the boundaries, or 2) their derivatives are zero at the boundaries. In the first instance, the natural transform to use is the *sine* transform, given by

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \qquad \text{sine transform} \tag{12.3.7}$$

where $f_j,\ j = 0, \ldots, N-1$ is the data array, and $f_0 \equiv 0$.

At first blush this appears to be simply the imaginary part of the discrete Fourier transform. However, the argument of the sine differs by a factor of two from the value that would make this so. The sine transform uses *sines only* as a complete set of functions in the interval from 0 to 2π, and, as we shall see, the cosine transform uses *cosines only*. By contrast, the normal FFT uses both sines and cosines, but only half as many of each. (See Figure 12.3.1.)

The expression (12.3.7) can be "force-fit" into a form that allows its calculation via the FFT. The idea is to extend the given function rightward past its last tabulated value. We extend the data to twice their length in such a way as to make them an *odd* function about $j = N$, with $f_N = 0$,

$$f_{2N-j} \equiv -f_j \qquad j = 0, \ldots, N-1 \tag{12.3.8}$$

Consider the FFT of this extended function:

$$F_k = \sum_{j=0}^{2N-1} f_j e^{2\pi ijk/(2N)} \tag{12.3.9}$$

The half of this sum from $j = N$ to $j = 2N-1$ can be rewritten with the substitution $j' = 2N - j$

$$\begin{aligned} \sum_{j=N}^{2N-1} f_j e^{2\pi ijk/(2N)} &= \sum_{j'=1}^{N} f_{2N-j'} e^{2\pi i(2N-j')k/(2N)} \\ &= -\sum_{j'=0}^{N-1} f_{j'} e^{-2\pi ij'k/(2N)} \end{aligned} \tag{12.3.10}$$

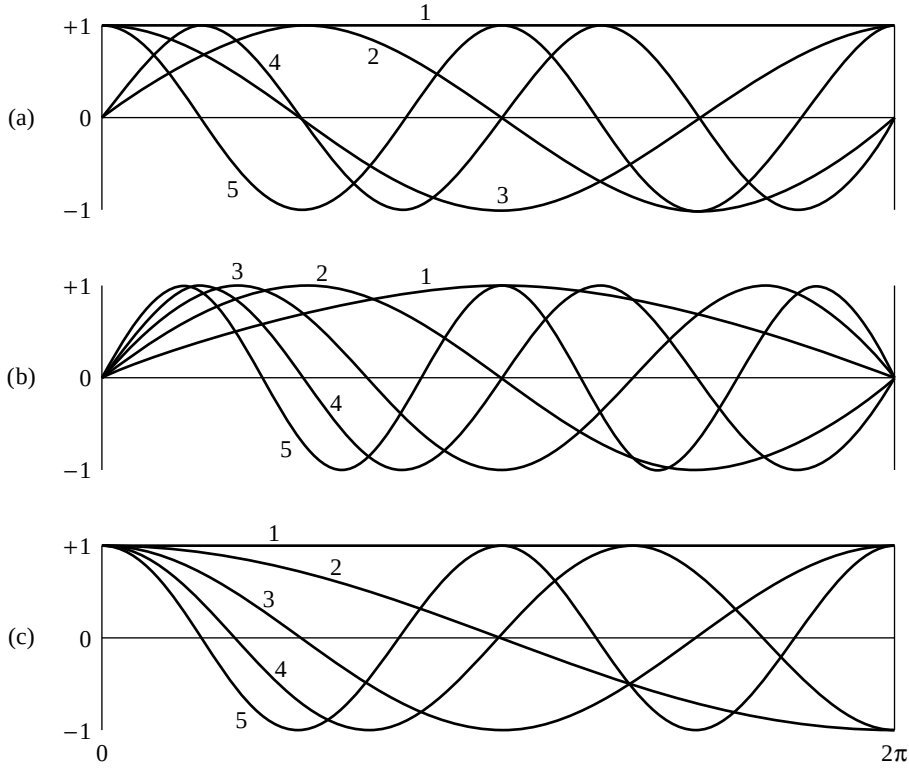


Figure 12.3.1. Basis functions used by the Fourier transform (a), sine transform (b), and cosine transform (c), are plotted. The first five basis functions are shown in each case. (For the Fourier transform, the real and imaginary parts of the basis functions are both shown.) While some basis functions occur in more than one transform, the basis sets are distinct. For example, the sine transform functions labeled (1), (3), (5) are not present in the Fourier basis. Any of the three sets can expand any function in the interval shown; however, the sine or cosine transform best expands functions matching the boundary conditions of the respective basis functions, namely zero function values for sine, zero derivatives for cosine.

so that

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} f_j \left[e^{2\pi i j k/(2N)} - e^{-2\pi i j k/(2N)} \right] \\ &= 2i \sum_{j=0}^{N-1} f_j \sin(\pi j k/N) \end{aligned} \tag{12.3.11}$$

Thus, up to a factor $2i$ we get the sine transform from the FFT of the extended function.

This method introduces a factor of two inefficiency into the computation by extending the data. This inefficiency shows up in the FFT output, which has zeros for the real part of every element of the transform. For a one-dimensional problem, the factor of two may be bearable, especially in view of the simplicity of the method. When we work with partial differential equations in two or three dimensions, though, the factor becomes four or eight, so efforts to eliminate the inefficiency are well rewarded.

From the original real data array f_j we will construct an auxiliary array y_j and apply to it the routine `realft`. The output will then be used to construct the desired transform. For the sine transform of data $f_j,\ j = 1, \ldots, N-1$, the auxiliary array is

$$\begin{aligned} y_0 &= 0 \\ y_j &= \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}) \qquad j = 1, \ldots, N-1 \end{aligned} \tag{12.3.12}$$

This array is of the same dimension as the original. Notice that the first term is symmetric about $j = N/2$ and the second is antisymmetric. Consequently, when `realft` is applied to y_j, the result has real parts R_k and imaginary parts I_k given by

$$\begin{aligned} R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} 2f_j \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \left[\sin\frac{(2k+1)j\pi}{N} - \sin\frac{(2k-1)j\pi}{N} \right] \\ &= F_{2k+1} - F_{2k-1} \end{aligned} \tag{12.3.13}$$

$$\begin{aligned} I_k &= \sum_{j=0}^{N-1} y_j \sin(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j - f_{N-j}) \frac{1}{2} \sin(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \sin(2\pi jk/N) \\ &= F_{2k} \end{aligned} \tag{12.3.14}$$

Therefore F_k can be determined as follows:

$$F_{2k} = I_k \qquad F_{2k+1} = F_{2k-1} + R_k \qquad k = 0, \ldots, (N/2-1) \tag{12.3.15}$$

The even terms of F_k are thus determined very directly. The odd terms require a recursion, the starting point of which follows from setting $k = 0$ in equation (12.3.15) and using $F_1 = -F_{-1}$:

$$F_1 = \frac{1}{2} R_0 \tag{12.3.16}$$

The implementing program is

```
      SUBROUTINE sinft(y,n)
      INTEGER n
      REAL y(n)
C     USES realft
        Calculates the sine transform of a set of n real-valued data points stored in array y(1:n).
        The number n must be a power of 2. On exit y is replaced by its transform. This program,
        without changes, also calculates the inverse sine transform, but in this case the output array
        should be multiplied by 2/n.
      INTEGER j
      REAL sum,y1,y2
      DOUBLE PRECISION theta,wi,wpi,wpr,
     *     wr,wtemp                          Double precision in the trigonometric recurrences.
      theta=3.141592653589793d0/dble(n)      Initialize the recurrence.
      wr=1.0d0
      wi=0.0d0
      wpr=-2.0d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      y(1)=0.0
      do 11 j=1,n/2
        wtemp=wr
        wr=wr*wpr-wi*wpi+wr                  Calculate the sine for the auxiliary array.
        wi=wi*wpr+wtemp*wpi+wi               The cosine is needed to continue the recurrence.
        y1=wi*(y(j+1)+y(n-j+1))              Construct the auxiliary array.
        y2=0.5*(y(j+1)-y(n-j+1))
        y(j+1)=y1+y2                         Terms j and N − j are related.
        y(n-j+1)=y1-y2
      enddo 11
      call realft(y,n,+1)                    Transform the auxiliary array.
      sum=0.0
      y(1)=0.5*y(1)                          Initialize the sum used for odd terms below.
      y(2)=0.0
      do 12 j=1,n-1,2
        sum=sum+y(j)
        y(j)=y(j+1)                          Even terms in the transform are determined directly.
        y(j+1)=sum                           Odd terms are determined by this running sum.
      enddo 12
      return
      END
```

The sine transform, curiously, is its own inverse. If you apply it twice, you get the original data, but multiplied by a factor of $N/2$.

The other common boundary condition for differential equations is that the derivative of the function is zero at the boundary. In this case the natural transform is the *cosine* transform. There are several possible ways of defining the transform. Each can be thought of as resulting from a different way of extending a given array to create an even array of double the length, and/or from whether the extended array contains $2N-1$, $2N$, or some other number of points. In practice, only two of the numerous possibilities are useful so we will restrict ourselves to just these two.

The first form of the cosine transform uses $N+1$ data points:

$$F_k = \frac{1}{2}[f_0 + (-1)^k f_N] + \sum_{j=1}^{N-1} f_j \cos(\pi jk/N) \tag{12.3.17}$$

It results from extending the given array to an even array about $j = N$, with

$$f_{2N-j} = f_j, \qquad j = 0, \ldots, N-1 \tag{12.3.18}$$

If you substitute this extended array into equation (12.3.9), and follow steps analogous to those leading up to equation (12.3.11), you will find that the Fourier transform is just twice the cosine transform (12.3.17). Another way of thinking about the formula (12.3.17) is to notice that it is the Chebyshev Gauss-Lobatto quadrature formula (see §4.5), often used in Clenshaw-Curtis adaptive quadrature (see §5.9, equation 5.9.4).

Once again the transform can be computed without the factor of two inefficiency. In this case the auxiliary function is

$$y_j = \frac{1}{2}(f_j + f_{N-j}) - \sin(j\pi/N)(f_j - f_{N-j}) \qquad j = 0, \ldots, N-1 \quad (12.3.19)$$

Instead of equation (12.3.15), `realft` now gives

$$F_{2k} = R_k \qquad F_{2k+1} = F_{2k-1} + I_k \qquad k = 0, \ldots, (N/2-1) \qquad (12.3.20)$$

The starting value for the recursion for odd k in this case is

$$F_1 = \frac{1}{2}(f_0 - f_N) + \sum_{j=1}^{N-1} f_j \cos(j\pi/N) \qquad (12.3.21)$$

This sum does not appear naturally among the R_k and I_k, and so we accumulate it during the generation of the array y_j.

Once again this transform is its own inverse, and so the following routine works for both directions of the transformation. Note that although this form of the cosine transform has $N+1$ input and output values, it passes an array only of length N to `realft`.

```
      SUBROUTINE cosft1(y,n)
      INTEGER n
      REAL y(n+1)
C     USES realft
        Calculates the cosine transform of a set y(1:n+1) of real-valued data points. The trans-
        formed data replace the original data in array y. n must be a power of 2. This program,
        without changes, also calculates the inverse cosine transform, but in this case the output
        array should be multiplied by 2/n.
      INTEGER j
      REAL sum,y1,y2
      DOUBLE PRECISION theta,wi,wpi,wpr,wr,wtemp        For trig. recurrences.
      theta=3.141592653589793d0/n                       Initialize the recurrence.
      wr=1.0d0
      wi=0.0d0
      wpr=-2.0d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      sum=0.5*(y(1)-y(n+1))
      y(1)=0.5*(y(1)+y(n+1))
      do 11 j=1,n/2-1                                   j=n/2 unnecessary since y(n/2+1) unchanged.
        wtemp=wr
        wr=wr*wpr-wi*wpi+wr                             Carry out the recurrence.
        wi=wi*wpr+wtemp*wpi+wi
        y1=0.5*(y(j+1)+y(n-j+1))                        Calculate the auxiliary function.
        y2=(y(j+1)-y(n-j+1))
        y(j+1)=y1-wi*y2                                 The values for j and N − j are related.
        y(n-j+1)=y1+wi*y2
        sum=sum+wr*y2                                   Carry along this sum for later use in unfolding the
      enddo 11                                              transform.
```

```
call realft(y,n,+1)
y(n+1)=y(2)
y(2)=sum
do 12 j=4,n,2
    sum=sum+y(j)
    y(j)=sum
enddo 12
return
END
```

Calculate the transform of the auxiliary function.

sum is the value of F_1 in equation (12.3.21).

Equation (12.3.20).

The second important form of the cosine transform is defined by

$$F_k = \sum_{j=0}^{N-1} f_j \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.22}$$

with inverse

$$f_j = \frac{2}{N} \sum_{k=0}^{N-1}{}' F_k \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.23}$$

Here the prime on the summation symbol means that the term for $k = 0$ has a coefficient of $\frac{1}{2}$ in front. This form arises by extending the given data, defined for $j = 0, \ldots, N-1$, to $j = N, \ldots, 2N-1$ in such a way that it is even about the point $N - \frac{1}{2}$ and periodic. (It is therefore also even about $j = -\frac{1}{2}$.) The form (12.3.23) is related to Gauss-Chebyshev quadrature (see equation 4.5.19), to Chebyshev approximation (§5.8, equation 5.8.7), and Clenshaw-Curtis quadrature (§5.9).

This form of the cosine transform is useful when solving differential equations on "staggered" grids, where the variables are centered midway between mesh points. It is also the standard form in the field of data compression and image processing.

The auxiliary function used in this case is similar to equation (12.3.19):

$$y_j = \frac{1}{2}(f_j + f_{N-j-1}) - \sin \frac{\pi(j+\frac{1}{2})}{N}(f_j - f_{N-j-1}) \qquad j = 0, \ldots, N-1 \tag{12.3.24}$$

Carrying out the steps similar to those used to get from (12.3.12) to (12.3.15), we find

$$F_{2k} = \cos \frac{\pi k}{N} R_k - \sin \frac{\pi k}{N} I_k \tag{12.3.25}$$

$$F_{2k-1} = \sin \frac{\pi k}{N} R_k + \cos \frac{\pi k}{N} I_k + F_{2k+1} \tag{12.3.26}$$

Note that equation (12.3.26) gives

$$F_{N-1} = \frac{1}{2} R_{N/2} \tag{12.3.27}$$

Thus the even components are found directly from (12.3.25), while the odd components are found by recursing (12.3.26) down from $k = N/2 - 1$, using (12.3.27) to start.

Since the transform is not self-inverting, we have to reverse the above steps to find the inverse. Here is the routine:

```
      SUBROUTINE cosft2(y,n,isign)
      INTEGER isign,n
      REAL y(n)
C     USES realft
```

Calculates the "staggered" cosine transform of a set y(1:n) of real-valued data points. The transformed data replace the original data in array y. n must be a power of 2. Set isign to $+1$ for a transform, and to -1 for an inverse transform. For an inverse transform, the output array should be multiplied by 2/n.

```
      INTEGER i
      REAL sum,sum1,y1,y2,ytemp
      DOUBLE PRECISION theta,wi,wi1,wpi,wpr,wr,wr1,wtemp,PI
```

Double precision for the trigonometric recurrences.

```
      PARAMETER (PI=3.141592653589793d0)
      theta=0.5d0*PI/n                    Initialize the recurrences.
      wr=1.0d0
      wi=0.0d0
      wr1=cos(theta)
      wi1=sin(theta)
      wpr=-2.0d0*wi1**2
      wpi=sin(2.d0*theta)
      if(isign.eq.1)then                  Forward transform.
        do 11 i=1,n/2
          y1=0.5*(y(i)+y(n-i+1))          Calculate the auxiliary function.
          y2=wi1*(y(i)-y(n-i+1))
          y(i)=y1+y2
          y(n-i+1)=y1-y2
          wtemp=wr1                       Carry out the recurrence.
          wr1=wr1*wpr-wi1*wpi+wr1
          wi1=wi1*wpr+wtemp*wpi+wi1
        enddo 11
        call realft(y,n,1)                Calculate the transform of the auxiliary function.
        do 12 i=3,n,2                     Even terms.
          wtemp=wr
          wr=wr*wpr-wi*wpi+wr
          wi=wi*wpr+wtemp*wpi+wi
          y1=y(i)*wr-y(i+1)*wi
          y2=y(i+1)*wr+y(i)*wi
          y(i)=y1
          y(i+1)=y2
        enddo 12
        sum=0.5*y(2)                      Initialize recurrence for odd terms with $\frac{1}{2}R_{N/2}$.
        do 13 i=n,2,-2                    Carry out recurrence for odd terms.
          sum1=sum
          sum=sum+y(i)
          y(i)=sum1
        enddo 13
      else if(isign.eq.-1)then            Inverse transform.
        ytemp=y(n)
        do 14 i=n,4,-2                    Form difference of odd terms.
          y(i)=y(i-2)-y(i)
        enddo 14
        y(2)=2.0*ytemp
        do 15 i=3,n,2                     Calculate $R_k$ and $I_k$.
          wtemp=wr
          wr=wr*wpr-wi*wpi+wr
          wi=wi*wpr+wtemp*wpi+wi
          y1=y(i)*wr+y(i+1)*wi
          y2=y(i+1)*wr-y(i)*wi
          y(i)=y1
          y(i+1)=y2
        enddo 15
        call realft(y,n,-1)
        do 16 i=1,n/2                     Invert auxiliary array.
          y1=y(i)+y(n-i+1)
```

```
        y2=(0.5/wi1)*(y(i)-y(n-i+1))
        y(i)=0.5*(y1+y2)
        y(n-i+1)=0.5*(y1-y2)
        wtemp=wr1
        wr1=wr1*wpr-wi1*wpi+wr1
        wi1=wi1*wpr+wtemp*wpi+wi1
    enddo 16
endif
return
END
```

An alternative way of implementing this algorithm is to form an auxiliary function by copying the even elements of f_j into the first $N/2$ locations, and the odd elements into the next $N/2$ elements in reverse order. However, it is not easy to implement the alternative algorithm without a temporary storage array and we prefer the above in-place algorithm.

Finally, we mention that there exist fast cosine transforms for small N that do not rely on an auxiliary function or use an FFT routine. Instead, they carry out the transform directly, often coded in hardware for fixed N of small dimension [1].

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §10–10.

Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burris, C.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849–863.

Hou, H.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455–1461 [see for additional references].

Hockney, R.W. 1971, in *Methods in Computational Physics*, vol. 9 (New York: Academic Press).

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329.

Clarke, R.J. 1985, *Transform Coding of Images*, (Reading, MA: Addison-Wesley).

Gonzalez, R.C., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C.H., and Fralick, S.C. 1977, *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009. [1]

12.4 FFT in Two or More Dimensions

Given a complex function $h(k_1, k_2)$ defined over the two-dimensional grid $0 \le k_1 \le N_1 - 1,\ 0 \le k_2 \le N_2 - 1$, we can define its two-dimensional discrete Fourier transform as a complex function $H(n_1, n_2)$, defined over the same grid,

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2/N_2)\ \exp(2\pi i k_1 n_1/N_1)\ h(k_1, k_2) \tag{12.4.1}$$

By pulling the "subscripts 2" exponential outside of the sum over k_1, or by reversing the order of summation and pulling the "subscripts 1" outside of the sum over k_2,

```
        y2=(0.5/wi1)*(y(i)-y(n-i+1))
        y(i)=0.5*(y1+y2)
        y(n-i+1)=0.5*(y1-y2)
        wtemp=wr1
        wr1=wr1*wpr-wi1*wpi+wr1
        wi1=wi1*wpr+wtemp*wpi+wi1
    enddo 16
endif
return
END
```

An alternative way of implementing this algorithm is to form an auxiliary function by copying the even elements of f_j into the first $N/2$ locations, and the odd elements into the next $N/2$ elements in reverse order. However, it is not easy to implement the alternative algorithm without a temporary storage array and we prefer the above in-place algorithm.

Finally, we mention that there exist fast cosine transforms for small N that do not rely on an auxiliary function or use an FFT routine. Instead, they carry out the transform directly, often coded in hardware for fixed N of small dimension [1].

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §10–10.

Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burris, C.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849–863.

Hou, H.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455–1461 [see for additional references].

Hockney, R.W. 1971, in *Methods in Computational Physics*, vol. 9 (New York: Academic Press).

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329.

Clarke, R.J. 1985, *Transform Coding of Images*, (Reading, MA: Addison-Wesley).

Gonzalez, R.C., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C.H., and Fralick, S.C. 1977, *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009. [1]

12.4 FFT in Two or More Dimensions

Given a complex function $h(k_1, k_2)$ defined over the two-dimensional grid $0 \le k_1 \le N_1 - 1,\ 0 \le k_2 \le N_2 - 1$, we can define its two-dimensional discrete Fourier transform as a complex function $H(n_1, n_2)$, defined over the same grid,

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2/N_2)\ \exp(2\pi i k_1 n_1/N_1)\ h(k_1, k_2) \tag{12.4.1}$$

By pulling the "subscripts 2" exponential outside of the sum over k_1, or by reversing the order of summation and pulling the "subscripts 1" outside of the sum over k_2,

we can see instantly that the two-dimensional FFT can be computed by taking one-dimensional FFTs sequentially on each index of the original function. Symbolically,

$$\begin{aligned} H(n_1, n_2) &= \text{FFT-on-index-1}\left(\text{FFT-on-index-2}\left[h(k_1, k_2)\right]\right) \\ &= \text{FFT-on-index-2}\left(\text{FFT-on-index-1}\left[h(k_1, k_2)\right]\right) \end{aligned} \tag{12.4.2}$$

For this to be practical, of course, both N_1 and N_2 should be some efficient length for an FFT, usually a power of 2. Programming a two-dimensional FFT, using (12.4.2) with a one-dimensional FFT routine, is a bit clumsier than it seems at first. Because the one-dimensional routine requires that its input be in consecutive order as a one-dimensional complex array, you find that you are endlessly copying things out of the multidimensional input array and then copying things back into it. This is not recommended technique. Rather, you should use a multidimensional FFT routine, such as the one we give below.

The generalization of (12.4.1) to more than two dimensions, say to L-dimensions, is evidently

$$\begin{aligned} H(n_1, \ldots, n_L) \equiv \sum_{k_L=0}^{N_L-1} \cdots \sum_{k_1=0}^{N_1-1} & \exp(2\pi i k_L n_L / N_L) \times \cdots \\ & \times \exp(2\pi i k_1 n_1 / N_1)\, h(k_1, \ldots, k_L) \end{aligned} \tag{12.4.3}$$

where n_1 and k_1 range from 0 to $N_1 - 1$, ... , n_L and k_L range from 0 to $N_L - 1$. How many calls to a one-dimensional FFT are in (12.4.3)? Quite a few! For each value of $k_1, k_2, \ldots, k_{L-1}$ you FFT to transform the L index. Then for each value of $k_1, k_2, \ldots, k_{L-2}$ and n_L you FFT to transform the $L - 1$ index. And so on. It is best to rely on someone else having done the bookkeeping for once and for all.

The inverse transforms of (12.4.1) or (12.4.3) are just what you would expect them to be: Change the i's in the exponentials to $-i$'s, and put an overall factor of $1/(N_1 \times \cdots \times N_L)$ in front of the whole thing. Most other features of multidimensional FFTs are also analogous to features already discussed in the one-dimensional case:

- Frequencies are arranged in wrap-around order in the transform, but now for each separate dimension.
- The input data are also treated as if they were wrapped around. If they are discontinuous across this periodic identification (in any dimension) then the spectrum will have some excess power at high frequencies because of the discontinuity. The fix, if you care, is to remove multidimensional linear trends.
- If you are doing spatial filtering and are worried about wrap-around effects, then you need to zero-pad all around the border of the multidimensional array. However, be sure to notice how costly zero-padding is in multidimensional transforms. If you use too thick a zero-pad, you are going to waste a *lot* of storage, especially in 3 or more dimensions!
- Aliasing occurs as always if sufficient bandwidth limiting does not exist along one or more of the dimensions of the transform.

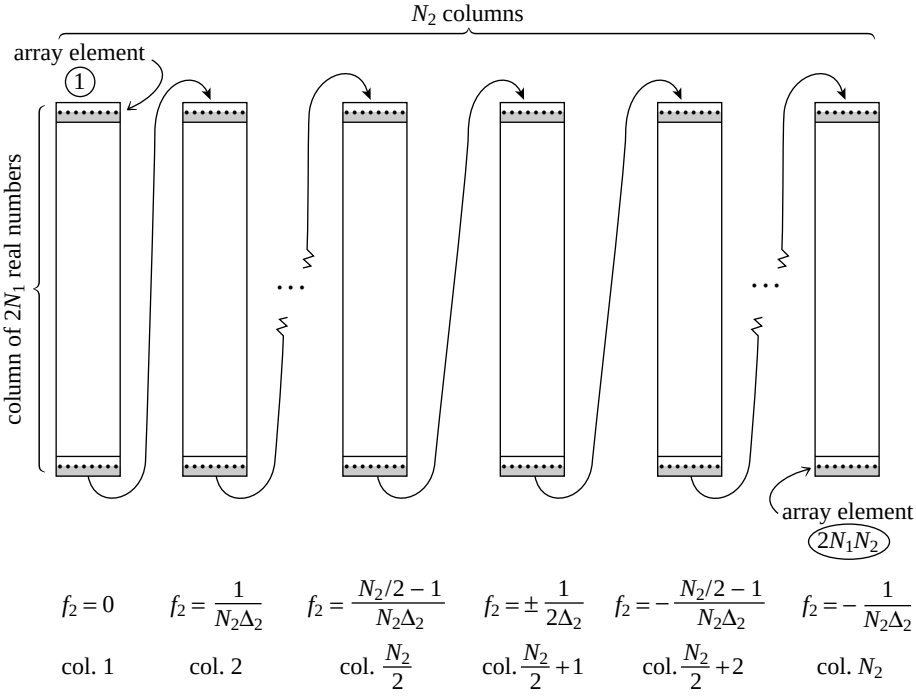


Figure 12.4.1. Storage arrangement of frequencies in the output $H(f_1, f_2)$ of a two-dimensional FFT. The input data is a two-dimensional $N_1 \times N_2$ array $h(t_1, t_2)$ (stored by columns of complex numbers). The output is also stored by complex columns. Each column corresponds to a particular value of f_2, as shown in the figure. Within each column, the arrangement of frequencies f_1 is exactly as shown in Figure 12.2.2 Δ_1 and Δ_2 are the sampling intervals in the 1 and 2 directions, respectively. The total number of (real) array elements is $2N_1N_2$. The program `fourn` can also do more than two dimensions, and the storage arrangement generalizes in the obvious way.

The routine `fourn` that we furnish herewith is a descendant of one written by N. M. Brenner. It requires as input (i) a scalar, telling the number of dimensions, e.g., 2; (ii) a vector, telling the length of the array in each dimension, e.g., (32,64). Note that these lengths *must all* be powers of 2, and are the numbers of *complex* values in each direction; (iii) the usual scalar equal to ± 1 indicating whether you want the transform or its inverse; and, finally (iv) the array of data.

A few words about the data array: `fourn` accesses it as a one-dimensional array of real numbers, of length equal to twice the product of the lengths of the L dimensions. It assumes that the array represents an L-dimensional complex array, in normal `FORTRAN` order. Normal `FORTRAN` order means: (i) each complex value occupies two sequential locations, real part followed by imaginary; (ii) the first subscript changes most rapidly as one goes through the array; the last subscript changes least rapidly; (iii) subscripts range from 1 to their maximum values ($N_1, N_2, \ldots, N_L$, respectively), rather than from 0 to $N_1 - 1,\ N_2 - 1, \ldots,\ N_L - 1$. Almost all failures to get `fourn` to work result from improper understanding of the above ordering of the data array, so take care! (Figure 12.4.1 illustrates the format of the output array.)

```
      SUBROUTINE fourn(data,nn,ndim,isign)
      INTEGER isign,ndim,nn(ndim)
      REAL data(*)
```

Replaces `data` by its `ndim`-dimensional discrete Fourier transform, if `isign` is input as 1. `nn(1:ndim)` is an integer array containing the lengths of each dimension (number of complex values), which MUST all be powers of 2. `data` is a real array of length twice the product of these lengths, in which the data are stored as in a multidimensional complex FORTRAN array. If `isign` is input as -1, `data` is replaced by its inverse transform times the product of the lengths of all dimensions.

```
      INTEGER i1,i2,i2rev,i3,i3rev,ibit,idim,ifp1,ifp2,ip1,ip2,
     *     ip3,k1,k2,n,nprev,nrem,ntot
      REAL tempi,tempr
      DOUBLE PRECISION theta,wi,wpi,wpr,wr,wtemp      Double precision for trigonometric re-
      ntot=1                                            currences.
      do 11 idim=1,ndim                 Compute total number of complex values.
        ntot=ntot*nn(idim)
      enddo 11
      nprev=1
      do 18 idim=1,ndim                 Main loop over the dimensions.
        n=nn(idim)
        nrem=ntot/(n*nprev)
        ip1=2*nprev
        ip2=ip1*n
        ip3=ip2*nrem
        i2rev=1
        do 14 i2=1,ip2,ip1              This is the bit-reversal section of the routine.
          if(i2.lt.i2rev)then
            do 13 i1=i2,i2+ip1-2,2
              do 12 i3=i1,ip3,ip2
                i3rev=i2rev+i3-i2
                tempr=data(i3)
                tempi=data(i3+1)
                data(i3)=data(i3rev)
                data(i3+1)=data(i3rev+1)
                data(i3rev)=tempr
                data(i3rev+1)=tempi
              enddo 12
            enddo 13
          endif
          ibit=ip2/2
1         if ((ibit.ge.ip1).and.(i2rev.gt.ibit)) then
            i2rev=i2rev-ibit
            ibit=ibit/2
          goto 1
          endif
          i2rev=i2rev+ibit
        enddo 14
        ifp1=ip1                        Here begins the Danielson-Lanczos section of the routine.
2       if(ifp1.lt.ip2)then
          ifp2=2*ifp1
          theta=isign*6.28318530717959d0/(ifp2/ip1)      Initialize for the trig. recur-
          wpr=-2.d0*sin(0.5d0*theta)**2                    rence.
          wpi=sin(theta)
          wr=1.d0
          wi=0.d0
          do 17 i3=1,ifp1,ip1
            do 16 i1=i3,i3+ip1-2,2
              do 15 i2=i1,ip3,ifp2
                k1=i2                   Danielson-Lanczos formula:
                k2=k1+ifp1
                tempr=sngl(wr)*data(k2)-sngl(wi)*data(k2+1)
                tempi=sngl(wr)*data(k2+1)+sngl(wi)*data(k2)
                data(k2)=data(k1)-tempr
                data(k2+1)=data(k1+1)-tempi
```

```
                data(k1)=data(k1)+tempr
                data(k1+1)=data(k1+1)+tempi
              enddo 15
            enddo 16
            wtemp=wr          Trigonometric recurrence.
            wr=wr*wpr-wi*wpi+wr
            wi=wi*wpr+wtemp*wpi+wi
          enddo 17
          ifp1=ifp2
        goto 2
        endif
        nprev=n*nprev
      enddo 18
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

12.5 Fourier Transforms of Real Data in Two and Three Dimensions

Two-dimensional FFTs are particularly important in the field of image processing. An image is usually represented as a two-dimensional array of pixel intensities, real (and usually positive) numbers. One commonly desires to filter high, or low, frequency spatial components from an image; or to convolve or deconvolve the image with some instrumental point spread function. Use of the FFT is by far the most efficient technique.

In three dimensions, a common use of the FFT is to solve Poisson's equation for a potential (e.g., electromagnetic or gravitational) on a three-dimensional lattice that represents the discretization of three-dimensional space. Here the source terms (mass or charge distribution) and the desired potentials are also real. In two and three dimensions, with large arrays, memory is often at a premium. It is therefore important to perform the FFTs, insofar as possible, on the data "in place." We want a routine with functionality similar to the multidimensional FFT routine `fourn` (§12.4), but which operates on real, not complex, input data. We give such a routine in this section. The development is analogous to that of §12.3 leading to the one-dimensional routine `realft`. (You might wish to review that material at this point, particularly equation 12.3.5.)

It is convenient to think of the independent variables $n_1, \ldots, n_L$ in equation (12.4.3) as representing an L-dimensional vector $\vec{n}$ in wave-number space, with values on the lattice of integers. The transform $H(n_1, \ldots, n_L)$ is then denoted $H(\vec{n})$.

It is easy to see that the transform $H(\vec{n})$ is periodic in each of its L dimensions. Specifically, if $\vec{P}_1, \vec{P}_2, \vec{P}_3, \ldots$ denote the vectors $(N_1, 0, 0, \ldots)$, $(0, N_2, 0, \ldots)$, $(0, 0, N_3, \ldots)$, and so forth, then

$$H(\vec{n} \pm \vec{P}_j) = H(\vec{n}) \qquad j = 1, \ldots, L \tag{12.5.1}$$

```
                data(k1)=data(k1)+tempr
                data(k1+1)=data(k1+1)+tempi
              enddo 15
            enddo 16
            wtemp=wr          Trigonometric recurrence.
            wr=wr*wpr-wi*wpi+wr
            wi=wi*wpr+wtemp*wpi+wi
          enddo 17
          ifp1=ifp2
        goto 2
        endif
        nprev=n*nprev
      enddo 18
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

12.5 Fourier Transforms of Real Data in Two and Three Dimensions

Two-dimensional FFTs are particularly important in the field of image processing. An image is usually represented as a two-dimensional array of pixel intensities, real (and usually positive) numbers. One commonly desires to filter high, or low, frequency spatial components from an image; or to convolve or deconvolve the image with some instrumental point spread function. Use of the FFT is by far the most efficient technique.

In three dimensions, a common use of the FFT is to solve Poisson's equation for a potential (e.g., electromagnetic or gravitational) on a three-dimensional lattice that represents the discretization of three-dimensional space. Here the source terms (mass or charge distribution) and the desired potentials are also real. In two and three dimensions, with large arrays, memory is often at a premium. It is therefore important to perform the FFTs, insofar as possible, on the data "in place." We want a routine with functionality similar to the multidimensional FFT routine `fourn` (§12.4), but which operates on real, not complex, input data. We give such a routine in this section. The development is analogous to that of §12.3 leading to the one-dimensional routine `realft`. (You might wish to review that material at this point, particularly equation 12.3.5.)

It is convenient to think of the independent variables $n_1, \ldots, n_L$ in equation (12.4.3) as representing an L-dimensional vector $\vec{n}$ in wave-number space, with values on the lattice of integers. The transform $H(n_1, \ldots, n_L)$ is then denoted $H(\vec{n})$.

It is easy to see that the transform $H(\vec{n})$ is periodic in each of its L dimensions. Specifically, if $\vec{P}_1, \vec{P}_2, \vec{P}_3, \ldots$ denote the vectors $(N_1, 0, 0, \ldots)$, $(0, N_2, 0, \ldots)$, $(0, 0, N_3, \ldots)$, and so forth, then

$$H(\vec{n} \pm \vec{P}_j) = H(\vec{n}) \qquad j = 1, \ldots, L \tag{12.5.1}$$

Equation (12.5.1) holds for any input data, real or complex. When the data is real, we have the additional symmetry

$$H(-\vec{n}) = H(\vec{n})^* \tag{12.5.2}$$

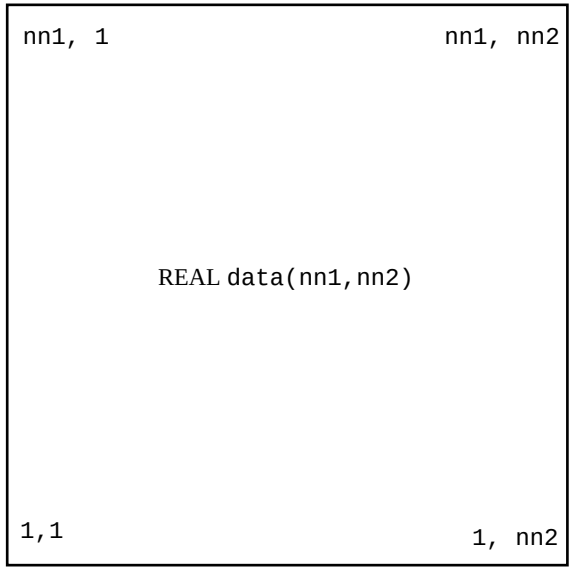
Equations (12.5.1) and (12.5.2) imply that the full transform can be trivially obtained from the subset of lattice values $\vec{n}$ that have

$$\begin{aligned} 0 &\le n_1 \le \frac{N_1}{2} \\ 0 &\le n_2 \le N_2 - 1 \\ &\cdots \\ 0 &\le n_L \le N_L - 1 \end{aligned} \tag{12.5.3}$$

In fact, this set of values is overcomplete, because there are additional symmetry relations among the transform values that have $n_1 = 0$ and $n_1 = N_1/2$. However these symmetries are complicated and their use becomes extremely confusing. Therefore, we will compute our FFT on the lattice subset of equation (12.5.3), even though this requires a small amount of extra storage for the answer, i.e., the transform is not *quite* "in place." (Although an in-place transform is in fact possible, we have found it virtually impossible to explain to any user how to unscramble its output, i.e., where to find the real and imaginary components of the transform at some particular frequency!)

Figure 12.5.1 shows the storage scheme that we will use for the input data and the output transform. The figure is specialized to the case of two dimensions, $L = 2$, but the generalization to higher dimensions is obvious. The input data is a two-dimensional real array of dimensions N_1 (called `nn1`) by N_2 (called `nn2`). Notice that the `FORTRAN` subscripts number from 1 to `nn1`, and not from 0 to $N_1 - 1$. The output spectrum is in two complex arrays, one two-dimensional and the other one-dimensional. The two-dimensional one, `spec`, has dimensions `nn1/2` by `nn2`. This is exactly half the size of the input data array; but since it is complex, it is the same amount of storage. In fact, `spec` will share storage with (and overwrite) the input data array. As the figure shows, `spec` contains those spectral components whose first component of frequency, f_1, ranges from zero to just short of the Nyquist frequency f_c. The full range of positive and negative second-component of frequencies, f_2, is stored, in wrap-around order (see §12.2), with negative frequencies shifted by exactly one period to put them "above" the positive frequencies, as the figure indicates. The figure also indicates how the additional $L - 1$ (here, one-) dimensional array `speq` stores only that single value of n_1 that corresponds to the Nyquist frequency, but all values of n_2, etc.

With this much introduction, the implementing procedure, called `rlft3`, is something of an anticlimax. The routine is written for the case of $L = 3$ dimensions, but (we will explain below) it can be used without modification for $L = 2$ also; and it is quite trivial to generalize it to larger L. Look at the innermost ("do $_{13}$") loop in the procedure, and you will see equation (12.3.5) implemented on the *first* transform index. The case of `i1=1` is coded separately, to account for the fact that `speq` is to be filled instead of `spec` (which is here called `data` since it shares storage with



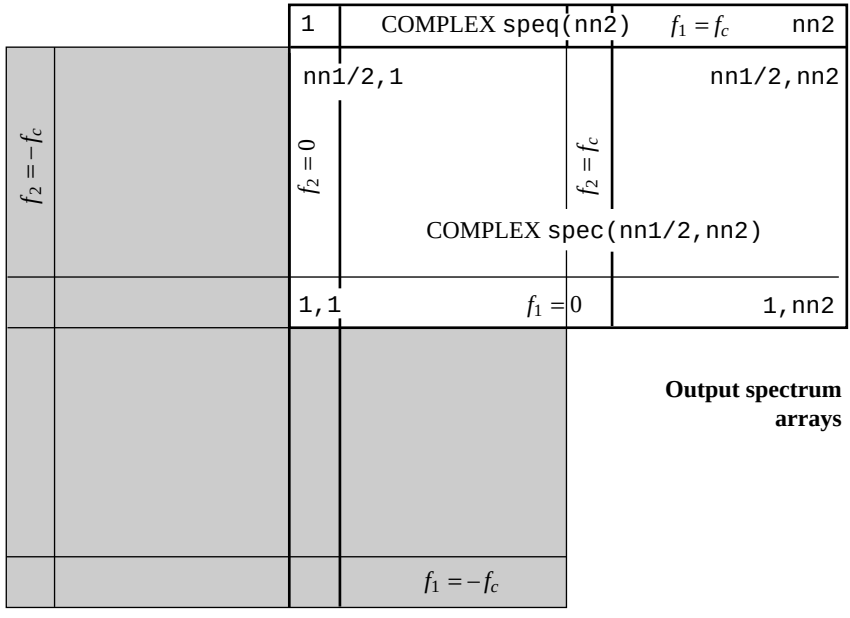


Figure 12.5.1. Input and output data arrangement for `rlft3` in the case of two-dimensional data. The input data array is a real, two-dimensional array. The output data array `spec` is a complex, two-dimensional array whose $(1,1)$ element contains the $f_1 = f_2 = 0$ spectral component; a complete set of f_2 values are stored in wrap-around order, while only positive f_1 values are stored (others being obtainable by symmetry). The output array `speq` contains components with f_1 equal to the Nyquist frequency.

the input array). The three enclosing do loops (indices i2, i1, and i3, from inside to outside) could in fact be done in any order — their actions all commute. We chose the order shown because of the following considerations: (i) i1 should not be the inner loop, because if it is, then the recurrence relations on wr and wi become burdensome. (ii) On virtual-memory machines, i3 should be the outer loop, because (with FORTRAN order of array storage) this results in the array data, which might be very large, being accessed in block sequential order.

Note that the work done in rlft3 is quite (logarithmically) small, compared to the associated complex FFT, fourn. For this reason, we allow ourselves the clarity of using FORTRAN complex arithmetic even when (as in the multiplications by c1 and c2) there are a few unnecessary operations. The routine rlft3 is based on an earlier routine by G.B. Rybicki.

```
      SUBROUTINE rlft3(data,speq,nn1,nn2,nn3,isign)
      INTEGER isign,nn1,nn2,nn3
      COMPLEX data(nn1/2,nn2,nn3),speq(nn2,nn3)
C     USES fourn
        Given a two- or three-dimensional real array data whose dimensions are nn1, nn2, nn3
        (where nn3 is 1 for the case of a two-dimensional array), this routine returns (for isign=1)
        the complex fast Fourier transform as two complex arrays: On output, data contains the
        zero and positive frequency values of the first frequency component, while speq contains
        the Nyquist critical frequency values of the first frequency component. Second (and third)
        frequency components are stored for zero, positive, and negative frequencies, in standard
        wrap-around order. For isign=-1, the inverse transform (times nn1*nn2*nn3/2 as a
        constant multiplicative factor) is performed, with output data (viewed as a real array)
        deriving from input data (viewed as complex) and speq. For inverse transforms on data
        not generated first by a forward transform, make sure the complex input data array satisfies
        property (12.5.2). The dimensions nn1, nn2, nn3 must always be integer powers of 2.
      INTEGER i1,i2,i3,j1,j2,j3,nn(3)
      DOUBLE PRECISION theta,wi,wpi,wpr,wr,wtemp
      COMPLEX c1,c2,h1,h2,w             Note that data is dimensioned as complex, its output
      c1=cmplx(0.5,0.0)                     format.
      c2=cmplx(0.0,-0.5*isign)
      theta=6.28318530717959d0/dble(isign*nn1)
      wpr=-2.0d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      nn(1)=nn1/2
      nn(2)=nn2
      nn(3)=nn3
      if(isign.eq.1)then                Case of forward transform.
        call fourn(data,nn,3,isign)     Here is where most all of the compute time is spent.
        do 12 i3=1,nn3                  Extend data periodically into speq.
          do 11 i2=1,nn2
            speq(i2,i3)=data(1,i2,i3)
          enddo 11
        enddo 12
      endif
      do 15 i3=1,nn3
        j3=1                            Zero frequency is its own reflection, otherwise locate cor-
        if (i3.ne.1) j3=nn3-i3+2            responding negative frequency in wrap-around order.
        wr=1.0d0                        Initialize trigonometric recurrence.
        wi=0.0d0
        do 14 i1=1,nn1/4+1
          j1=nn1/2-i1+2
          do 13 i2=1,nn2
            j2=1
            if (i2.ne.1) j2=nn2-i2+2
            if(i1.eq.1)then             Equation (12.3.5).
              h1=c1*(data(1,i2,i3)+conjg(speq(j2,j3)))
              h2=c2*(data(1,i2,i3)-conjg(speq(j2,j3)))
```

Figure 12.5.2. (a) A two-dimensional image with intensities either purely black or purely white. (b) The same image, after it has been low-pass filtered using `rlft3`. Regions with fine-scale features become gray.

```
                data(1,i2,i3)=h1+h2
                speq(j2,j3)=conjg(h1-h2)
            else
                h1=c1*(data(i1,i2,i3)+conjg(data(j1,j2,j3)))
                h2=c2*(data(i1,i2,i3)-conjg(data(j1,j2,j3)))
                data(i1,i2,i3)=h1+w*h2
                data(j1,j2,j3)=conjg(h1-w*h2)
            endif
        enddo 13
        wtemp=wr                    Do the recurrence.
        wr=wr*wpr-wi*wpi+wr
        wi=wi*wpr+wtemp*wpi+wi
        w=cmplx(sngl(wr),sngl(wi))
    enddo 14
enddo 15
if(isign.eq.-1)then                 Case of reverse transform.
    call fourn(data,nn,3,isign)
endif
return
END
```

We now give some fragments from notional calling programs, to clarify the use of `rlft3` for two- and three-dimensional data. Note that the routine does not actually distinguish between two and three dimensions; two is treated like three, but with the third dimension having length 1. Since the third dimension is the outer loop, almost no inefficiency is introduced.

The first program fragment FFTs a two-dimensional data array, allows for some processing on it, e.g., filtering, and then takes the inverse transform. Figure 12.5.2 shows an example of the use of this kind of code: A sharp image becomes blurry when its high-frequency spatial components are suppressed by the factor (here) $\max\,(1 - 6f^2/f_c^2, 0)$. The second program example illustrates a three-dimensional transform, where the three dimensions have different lengths. The third program example is an example of convolution, as it might occur in a program to compute the potential generated by a three-dimensional distribution of sources.

```
      PROGRAM exmpl1
          This fragment shows how one might filter a 256 by 256 digital image.
      INTEGER N1,N2,N3
      PARAMETER (N1=256,N2=256,N3=1)  Note that the third component must be set to 1.
C     USES rlft3
      REAL data(N1,N2)
      COMPLEX spec(N1/2,N2),speq(N2)
      EQUIVALENCE (data,spec)
C     ...                             Here the image would be loaded into data.
      call rlft3(data,speq,N1,N2,N3,1)
C     ...                             Here the arrays spec and speq would be multiplied by a suit-
      call rlft3(data,speq,N1,N2,N3,-1)       able filter function (of frequency).
C     ...                             Here the filtered image would be unloaded from data.
      END

      PROGRAM exmpl2
          This fragment shows how one might FFT a real three-dimensional array of size 32 by 64
          by 16.
      INTEGER N1,N2,N3
      PARAMETER (N1=32,N2=64,N3=16)
C     USES rlft3
      REAL data(N1,N2,N3)
      COMPLEX spec(N1/2,N2,N3),speq(N2,N3)
      EQUIVALENCE (data,spec)
C     ...                             Here load data.
      call rlft3(data,speq,N1,N2,N3,1)
C     ...                             Here unload spec and speq.
      END

      PROGRAM exmpl3
          This fragment shows how one might convolve two real, three-dimensional arrays of size 32
          by 32 by 32, replacing the first array by the result.
      INTEGER N
      PARAMETER (N=32)
C     USES rlft3
      INTEGER j
      REAL fac,data1(N,N,N),data2(N,N,N)
      COMPLEX spec1(N/2,N,N),speq1(N,N),spec2(N/2,N,N),speq2(N,N),
     *     zpec1(N*N*N/2),zpeq1(N*N),zpec2(N*N*N/2),zpeq2(N*N)
      EQUIVALENCE (data1,spec1,zpec1), (data2,spec2,zpec2),
     *     (speq1,zpeq1), (speq2,zpeq2)
C     ...
      call rlft3(data1,speq1,N,N,N,1)       FFT both input arrays.
      call rlft3(data2,speq2,N,N,N,1)
      fac=2./(N*N*N)                    Factor needed to get normalized inverse.
      do 11 j=1,N*N*N/2                 The sole purpose of the zpecs and zpeqs is to make
          zpec1(j)=fac*zpec1(j)*zpec2(j)    this a single do-loop instead of three-nested ones.
      enddo 11
      do 12 j=1,N*N
          zpeq1(j)=fac*zpeq1(j)*zpeq2(j)
      enddo 12
      call rlft3(data1,speq1,N,N,N,-1)  Inverse FFT the product of the two FFTs.
C     ...
      END
```

To extend `rlft3` to four dimensions, you simply add an additional (outer) nested `do` loop in `i4`, analogous to the present `i3`. (Modifying the routine to do an *arbitrary* number of dimensions, as in `fourn`, is a good programming exercise for the reader.)

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Swartztrauber, P. N. 1986, *Mathematics of Computation*, vol. 47, pp. 323–346.

12.6 External Storage or Memory-Local FFTs

Sometime in your life, you might have to compute the Fourier transform of a *really large* data set, larger than the size of your computer's physical memory. In such a case, the data will be stored on some external medium, such as magnetic or optical tape or disk. Needed is an algorithm that makes some manageable number of sequential passes through the external data, processing it on the fly and outputting intermediate results to other external media, which can be read on subsequent passes.

In fact, an algorithm of just this description was developed by Singleton [1] very soon after the discovery of the FFT. The algorithm requires four sequential storage devices, each capable of holding half of the input data. The first half of the input data is initially on one device, the second half on another.

Singleton's algorithm is based on the observation that it is possible to bit-reverse 2^M values by the following sequence of operations: On the first pass, values are read alternately from the two input devices, and written to a single output device (until it holds half the data), and then to the other output device. On the second pass, the output devices become input devices, and vice versa. Now, we copy *two* values from the first device, then *two* values from the second, writing them (as before) first to fill one output device, then to fill a second. Subsequent passes read 4, 8, etc., input values at a time. After completion of pass $M - 1$, the data are in bit-reverse order.

Singleton's next observation is that it is possible to alternate the passes of essentially this bit-reversal technique with passes that implement one stage of the Danielson-Lanczos combination formula (12.2.3). The scheme, roughly, is this: One starts as before with half the input data on one device, half on another. In the first pass, one complex value is read from each input device. Two combinations are formed, and one is written to each of two output devices. After this "computing" pass, the devices are rewound, and a "permutation" pass is performed, where groups of values are read from the first input device and alternately written to the first and second output devices; when the first input device is exhausted, the second is similarly processed. This sequence of computing and permutation passes is repeated $M - K - 1$ times, where 2^K is the size of internal buffer available to the program. The second phase of the computation consists of a final K computation passes. What distinguishes the second phase from the first is that, now, the permutations are local enough to do in place during the computation. There are thus no separate permutation passes in the second phase. In all, there are $2M - K - 2$ passes through the data.

Here is an implementation of Singleton's algorithm, based on [1]:

```
      SUBROUTINE fourfs(iunit,nn,ndim,isign)
      INTEGER ndim,nn(ndim),isign,iunit(4),KBF
      PARAMETER (KBF=128)
C     USES fourew
```

One- or multi-dimensional Fourier transform of a large data set stored on external media. On input, ndim is the number of dimensions, and nn(1:ndim) contains the lengths of each dimension (number of complex values), which must be powers of two. iunit(1:4) contains the unit numbers of 4 sequential files, each large enough to hold half of the data. The four units must be opened for FORTRAN unformatted access. The input data must be in FORTRAN normal order, with its first half stored on unit iunit(1), its second half on iunit(2), in unformatted form, with KBF real numbers per record. isign should be set to 1 for the Fourier transform, to -1 for its inverse. On output, values in the array iunit may have been permuted; the first half of the result is stored on iunit(3), the second half on iunit(4). N.B.: For ndim > 1, the output is stored by rows, i.e., *not* in FORTRAN normal order; in other words, the output is the transpose of that which would have been produced by routine fourn.

```
      INTEGER j,j12,jk,k,kk,n,mm,kc,kd,ks,kr,nr,ns,nv,jx,
     *     mate(4),na,nb,nc,nd
      REAL tempr,tempi,afa(KBF),afb(KBF),afc(KBF)
      DOUBLE PRECISION wr,wi,wpr,wpi,wtemp,theta
      SAVE mate
      DATA mate /2,1,4,3/
```

12.6 External Storage or Memory-Local FFTs

Sometime in your life, you might have to compute the Fourier transform of a *really large* data set, larger than the size of your computer's physical memory. In such a case, the data will be stored on some external medium, such as magnetic or optical tape or disk. Needed is an algorithm that makes some manageable number of sequential passes through the external data, processing it on the fly and outputting intermediate results to other external media, which can be read on subsequent passes.

In fact, an algorithm of just this description was developed by Singleton [1] very soon after the discovery of the FFT. The algorithm requires four sequential storage devices, each capable of holding half of the input data. The first half of the input data is initially on one device, the second half on another.

Singleton's algorithm is based on the observation that it is possible to bit-reverse 2^M values by the following sequence of operations: On the first pass, values are read alternately from the two input devices, and written to a single output device (until it holds half the data), and then to the other output device. On the second pass, the output devices become input devices, and vice versa. Now, we copy *two* values from the first device, then *two* values from the second, writing them (as before) first to fill one output device, then to fill a second. Subsequent passes read 4, 8, etc., input values at a time. After completion of pass $M - 1$, the data are in bit-reverse order.

Singleton's next observation is that it is possible to alternate the passes of essentially this bit-reversal technique with passes that implement one stage of the Danielson-Lanczos combination formula (12.2.3). The scheme, roughly, is this: One starts as before with half the input data on one device, half on another. In the first pass, one complex value is read from each input device. Two combinations are formed, and one is written to each of two output devices. After this "computing" pass, the devices are rewound, and a "permutation" pass is performed, where groups of values are read from the first input device and alternately written to the first and second output devices; when the first input device is exhausted, the second is similarly processed. This sequence of computing and permutation passes is repeated $M - K - 1$ times, where 2^K is the size of internal buffer available to the program. The second phase of the computation consists of a final K computation passes. What distinguishes the second phase from the first is that, now, the permutations are local enough to do in place during the computation. There are thus no separate permutation passes in the second phase. In all, there are $2M - K - 2$ passes through the data.

Here is an implementation of Singleton's algorithm, based on [1]:

```
      SUBROUTINE fourfs(iunit,nn,ndim,isign)
      INTEGER ndim,nn(ndim),isign,iunit(4),KBF
      PARAMETER (KBF=128)
C     USES fourew
```

One- or multi-dimensional Fourier transform of a large data set stored on external media. On input, ndim is the number of dimensions, and nn(1:ndim) contains the lengths of each dimension (number of complex values), which must be powers of two. iunit(1:4) contains the unit numbers of 4 sequential files, each large enough to hold half of the data. The four units must be opened for FORTRAN unformatted access. The input data must be in FORTRAN normal order, with its first half stored on unit iunit(1), its second half on iunit(2), in unformatted form, with KBF real numbers per record. isign should be set to 1 for the Fourier transform, to -1 for its inverse. On output, values in the array iunit may have been permuted; the first half of the result is stored on iunit(3), the second half on iunit(4). N.B.: For ndim > 1, the output is stored by rows, i.e., *not* in FORTRAN normal order; in other words, the output is the transpose of that which would have been produced by routine fourn.

```
      INTEGER j,j12,jk,k,kk,n,mm,kc,kd,ks,kr,nr,ns,nv,jx,
     *     mate(4),na,nb,nc,nd
      REAL tempr,tempi,afa(KBF),afb(KBF),afc(KBF)
      DOUBLE PRECISION wr,wi,wpr,wpi,wtemp,theta
      SAVE mate
      DATA mate /2,1,4,3/
```

```
      n=1
      do 11 j=1,ndim
          n=n*nn(j)
          if (nn(j).le.1)
*              pause 'invalid dimension or wrong ndim in fourfs'
      enddo 11
      nv=ndim
      jk=nn(nv)
      mm=n
      ns=n/KBF
      nr=ns/2
      kc=0
      kd=KBF/2
      ks=n
      call fourew(iunit,na,nb,nc,nd)
        The first phase of the transform starts here.
1     continue                          Start of the computing pass.
          theta=3.141592653589793d0/(isign*n/mm)
          wpr=-2.d0*sin(0.5d0*theta)**2
          wpi=sin(theta)
          wr=1.d0
          wi=0.d0
          mm=mm/2
          do 13 j12=1,2
              kr=0
2             continue
                  read (iunit(na)) (afa(jx),jx=1,KBF)
                  read (iunit(nb)) (afb(jx),jx=1,KBF)
                  do 12 j=1,KBF,2
                      tempr=sngl(wr)*afb(j)-sngl(wi)*afb(j+1)
                      tempi=sngl(wi)*afb(j)+sngl(wr)*afb(j+1)
                      afb(j)=afa(j)-tempr
                      afa(j)=afa(j)+tempr
                      afb(j+1)=afa(j+1)-tempi
                      afa(j+1)=afa(j+1)+tempi
                  enddo 12
                  kc=kc+kd
                  if (kc.eq.mm) then
                      kc=0
                      wtemp=wr
                      wr=wr*wpr-wi*wpi+wr
                      wi=wi*wpr+wtemp*wpi+wi
                  endif
                  write (iunit(nc)) (afa(jx),jx=1,KBF)
                  write (iunit(nd)) (afb(jx),jx=1,KBF)
              kr=kr+1
              if (kr.lt.nr) goto 2
              if(j12.eq.1.and.ks.ne.n.and.ks.eq.KBF) then
                  na=mate(na)
                  nb=na
              endif
              if (nr.eq.0) goto 3
          enddo 13
3         call fourew(iunit,na,nb,nc,nd)        Start of the permutation pass.
          jk=jk/2
4         if (jk.eq.1) then
              mm=n
              nv=nv-1
              jk=nn(nv)
          goto 4
          endif
          ks=ks/2
          if (ks.gt.KBF) then
              do 16 j12=1,2
```

```
            do 15 kr=1,ns,ks/KBF
                do 14 k=1,ks,KBF
                    read (iunit(na)) (afa(jx),jx=1,KBF)
                    write (iunit(nc)) (afa(jx),jx=1,KBF)
                enddo 14
                nc=mate(nc)
            enddo 15
            na=mate(na)
        enddo 16
        call fourew(iunit,na,nb,nc,nd)
        goto 1
    else if (ks.eq.KBF) then
        nb=na
        goto 1
    endif
continue
j=1
```

The second phase of the transform starts here. Now, the remaining permutations are sufficiently local to be done in place.

```
5   continue
    theta=3.141592653589793d0/(isign*n/mm)
    wpr=-2.d0*sin(0.5d0*theta)**2
    wpi=sin(theta)
    wr=1.d0
    wi=0.d0
    mm=mm/2
    ks=kd
    kd=kd/2
    do 18 j12=1,2
        do 17 kr=1,ns
            read (iunit(na)) (afc(jx),jx=1,KBF)
            kk=1
            k=ks+1
6           continue
                tempr=sngl(wr)*afc(kk+ks)-sngl(wi)*afc(kk+ks+1)
                tempi=sngl(wi)*afc(kk+ks)+sngl(wr)*afc(kk+ks+1)
                afa(j)=afc(kk)+tempr
                afb(j)=afc(kk)-tempr
                afa(j+1)=afc(kk+1)+tempi
                afb(j+1)=afc(kk+1)-tempi
                j=j+2
                kk=kk+2
            if (kk.lt.k) goto 6
            kc=kc+kd
            if (kc.eq.mm) then
                kc=0
                wtemp=wr
                wr=wr*wpr-wi*wpi+wr
                wi=wi*wpr+wtemp*wpi+wi
            endif
            kk=kk+ks
            if (kk.le.KBF) then
                k=kk+ks
                goto 6
            endif
            if (j.gt.KBF) then
                write (iunit(nc)) (afa(jx),jx=1,KBF)
                write (iunit(nd)) (afb(jx),jx=1,KBF)
                j=1
            endif
        enddo 17
        na=mate(na)
    enddo 18
    call fourew(iunit,na,nb,nc,nd)
```

```
        jk=jk/2
      if (jk.gt.1) goto 5
      mm=n
7     if (nv.gt.1) then
        nv=nv-1
        jk=nn(nv)
        if (jk.eq.1) goto 7
        goto 5
      endif
      return
      END



      SUBROUTINE fourew(iunit,na,nb,nc,nd)
      INTEGER na,nb,nc,nd,iunit(4),ii
        Utility used by fourfs. Rewinds and renumbers the four files.
      do 11 ii=1,4
        rewind(unit=iunit(ii))
      enddo 11
      ii=iunit(2)
      iunit(2)=iunit(4)
      iunit(4)=ii
      ii=iunit(1)
      iunit(1)=iunit(3)
      iunit(3)=ii
      na=3
      nb=4
      nc=1
      nd=2
      return
      END
```

For one-dimensional data, Singleton's algorithm produces output in exactly the same order as a standard FFT (e.g., `four1`). For multidimensional data, the output is the *transpose* of the conventional arrangement (e.g., the output of `fourn`). This peculiarity, which is intrinsic to the method, is generally only a minor inconvenience. For convolutions, one simply computes the component-by-component product of two transforms in their nonstandard arrangement, and then does an inverse transform on the result. Note that, if the lengths of the different dimensions are not all the same, then you must reverse the order of the values in `nn(1:ndim)` (thus giving the transpose dimensions) before performing the inverse transform. Note also that, just like `fourn`, performing a transform and then an inverse results in multiplying the original data by the product of the lengths of all dimensions.

We leave it as an exercise for the reader to figure out how to reorder `fourfs`'s output into normal order, taking additional passes through the externally stored data. We doubt that such reordering is ever really needed.

You will likely want to modify `fourfs` to fit your particular application. For example, as written, `KBF` $\equiv 2^K$ plays the dual role of being the size of the internal buffers, and the record size of the unformatted reads and writes. The latter role limits its size to that allowed by your machine's I/O facility. It is a simple matter to perform multiple reads for a much larger `KBF`, thus reducing the number of passes by a few.

Another modification of `fourfs` would be for the case where your virtual memory machine has sufficient address space, but not sufficient physical memory, to do an efficient FFT by the conventional algorithm (whose memory references are extremely nonlocal). In that case, you will need to replace the reads, writes, and rewinds by mappings of the arrays `afa`, `afb`, and `afc` into your address space. In other words, these arrays are replaced by references to a single data array, with offsets that get modified wherever `fourfs` performs an I/O operation. The resulting algorithm will have its memory references local within blocks of size `KBF`. Execution speed is thereby sometimes increased enormously, albeit at the cost of requiring twice as much virtual memory as an in-place FFT.

CITED REFERENCES AND FURTHER READING:

Singleton, R.C. 1967, *IEEE Transactions on Audio and Electroacoustics*, vol. AU-15, pp. 91–97. [1]

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

Chapter 13. Fourier and Spectral Applications

13.0 Introduction

Fourier methods have revolutionized fields of science and engineering, from radio astronomy to medical imaging, from seismology to spectroscopy. In this chapter, we present some of the basic applications of Fourier and spectral methods that have made these revolutions possible.

Say the word "Fourier" to a numericist, and the response, as if by Pavlovian conditioning, will likely be "FFT." Indeed, the wide application of Fourier methods must be credited principally to the existence of the fast Fourier transform. Better mousetraps stand aside: If you speed up *any* nontrivial algorithm by a factor of a million or so, the world will beat a path towards finding useful applications for it. The most direct applications of the FFT are to the convolution or deconvolution of data (§13.1), correlation and autocorrelation (§13.2), optimal filtering (§13.3), power spectrum estimation (§13.4), and the computation of Fourier integrals (§13.9).

As important as they are, however, FFT methods are not the be-all and end-all of spectral analysis. Section 13.5 is a brief introduction to the field of time-domain digital filters. In the spectral domain, one limitation of the FFT is that it always represents a function's Fourier transform as a polynomial in $z = \exp(2\pi i f \Delta)$ (cf. equation 12.1.7). Sometimes, processes have spectra whose shapes are not well represented by this form. An alternative form, which allows the spectrum to have poles in z, is used in the techniques of linear prediction (§13.6) and maximum entropy spectral estimation (§13.7).

Another significant limitation of all FFT methods is that they require the input data to be sampled at evenly spaced intervals. For irregularly or incompletely sampled data, other (albeit slower) methods are available, as discussed in §13.8.

So-called wavelet methods inhabit a representation of function space that is neither in the temporal, nor in the spectral, domain, but rather something in-between. Section 13.10 is an introduction to this subject. Finally §13.11 is an excursion into numerical use of the Fourier sampling theorem.

13.1 Convolution and Deconvolution Using the FFT

We have defined the *convolution* of two functions for the continuous case in equation (12.0.8), and have given the *convolution theorem* as equation (12.0.9). The theorem says that the Fourier transform of the convolution of two functions is equal to the product of their individual Fourier transforms. Now, we want to deal with the discrete case. We will mention first the context in which convolution is a useful procedure, and then discuss how to compute it efficiently using the FFT.

The convolution of two functions $r(t)$ and $s(t)$, denoted $r * s$, is mathematically equal to their convolution in the opposite order, $s * r$. Nevertheless, in most applications the two functions have quite different meanings and characters. One of the functions, say s, is typically a signal or data stream, which goes on indefinitely in time (or in whatever the appropriate independent variable may be). The other function r is a "response function," typically a peaked function that falls to zero in both directions from its maximum. The effect of convolution is to smear the signal $s(t)$ in time according to the recipe provided by the response function $r(t)$, as shown in Figure 13.1.1. In particular, a spike or delta-function of unit area in s which occurs at some time t_0 is supposed to be smeared into the shape of the response function itself, but translated from time 0 to time t_0 as $r(t - t_0)$.

In the discrete case, the signal $s(t)$ is represented by its sampled values at equal time intervals s_j. The response function is also a discrete set of numbers r_k, with the following interpretation: r_0 tells what multiple of the input signal in one channel (one particular value of j) is copied into the identical output channel (same value of j); r_1 tells what multiple of input signal in channel j is additionally copied into output channel $j + 1$; r_{-1} tells the multiple that is copied into channel $j - 1$; and so on for both positive and negative values of k in r_k. Figure 13.1.2 illustrates the situation.

Example: a response function with $r_0 = 1$ and all other r_k's equal to zero is just the identity filter: convolution of a signal with this response function gives identically the signal. Another example is the response function with $r_{14} = 1.5$ and all other r_k's equal to zero. This produces convolved output that is the input signal multiplied by 1.5 and delayed by 14 sample intervals.

Evidently, we have just described in words the following definition of discrete convolution with a response function of finite duration M:

$$(r * s)_j \equiv \sum_{k=-M/2+1}^{M/2} s_{j-k}\, r_k \tag{13.1.1}$$

If a discrete response function is nonzero only in some range $-M/2 < k \le M/2$, where M is a sufficiently large even integer, then the response function is called a *finite impulse response (FIR)*, and its *duration* is M. (Notice that we are defining M as the number of nonzero *values* of r_k; these values span a time interval of $M - 1$ sampling times.) In most practical circumstances the case of finite M is the case of interest, either because the response really has a finite duration, or because we choose to truncate it at some point and approximate it by a finite-duration response function.

The *discrete convolution theorem* is this: If a signal s_j is *periodic* with period N, so that it is completely determined by the N values $s_0, \ldots, s_{N-1}$, then its

13.1 Convolution and Deconvolution Using the FFT

We have defined the *convolution* of two functions for the continuous case in equation (12.0.8), and have given the *convolution theorem* as equation (12.0.9). The theorem says that the Fourier transform of the convolution of two functions is equal to the product of their individual Fourier transforms. Now, we want to deal with the discrete case. We will mention first the context in which convolution is a useful procedure, and then discuss how to compute it efficiently using the FFT.

The convolution of two functions $r(t)$ and $s(t)$, denoted $r * s$, is mathematically equal to their convolution in the opposite order, $s * r$. Nevertheless, in most applications the two functions have quite different meanings and characters. One of the functions, say s, is typically a signal or data stream, which goes on indefinitely in time (or in whatever the appropriate independent variable may be). The other function r is a "response function," typically a peaked function that falls to zero in both directions from its maximum. The effect of convolution is to smear the signal $s(t)$ in time according to the recipe provided by the response function $r(t)$, as shown in Figure 13.1.1. In particular, a spike or delta-function of unit area in s which occurs at some time t_0 is supposed to be smeared into the shape of the response function itself, but translated from time 0 to time t_0 as $r(t - t_0)$.

In the discrete case, the signal $s(t)$ is represented by its sampled values at equal time intervals s_j. The response function is also a discrete set of numbers r_k, with the following interpretation: r_0 tells what multiple of the input signal in one channel (one particular value of j) is copied into the identical output channel (same value of j); r_1 tells what multiple of input signal in channel j is additionally copied into output channel $j + 1$; r_{-1} tells the multiple that is copied into channel $j - 1$; and so on for both positive and negative values of k in r_k. Figure 13.1.2 illustrates the situation.

Example: a response function with $r_0 = 1$ and all other r_k's equal to zero is just the identity filter: convolution of a signal with this response function gives identically the signal. Another example is the response function with $r_{14} = 1.5$ and all other r_k's equal to zero. This produces convolved output that is the input signal multiplied by 1.5 and delayed by 14 sample intervals.

Evidently, we have just described in words the following definition of discrete convolution with a response function of finite duration M:

$$(r * s)_j \equiv \sum_{k=-M/2+1}^{M/2} s_{j-k}\, r_k \tag{13.1.1}$$

If a discrete response function is nonzero only in some range $-M/2 < k \le M/2$, where M is a sufficiently large even integer, then the response function is called a *finite impulse response (FIR)*, and its *duration* is M. (Notice that we are defining M as the number of nonzero *values* of r_k; these values span a time interval of $M - 1$ sampling times.) In most practical circumstances the case of finite M is the case of interest, either because the response really has a finite duration, or because we choose to truncate it at some point and approximate it by a finite-duration response function.

The *discrete convolution theorem* is this: If a signal s_j is *periodic* with period N, so that it is completely determined by the N values $s_0, \ldots, s_{N-1}$, then its

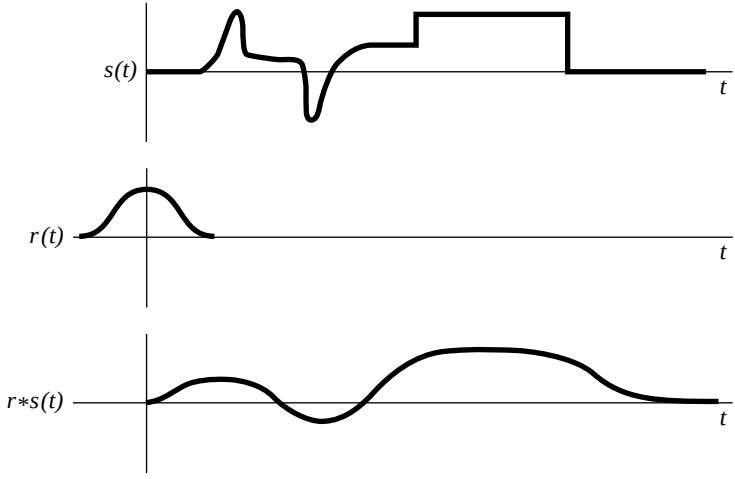


Figure 13.1.1. Example of the convolution of two functions. A signal $s(t)$ is convolved with a response function $r(t)$. Since the response function is broader than some features in the original signal, these are "washed out" in the convolution. In the absence of any additional noise, the process can be reversed by deconvolution.

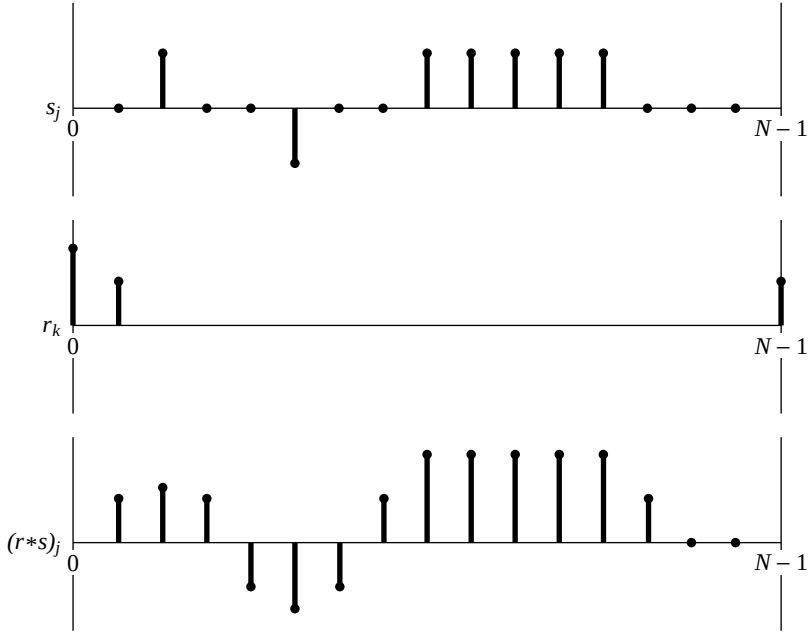


Figure 13.1.2. Convolution of discretely sampled functions. Note how the response function for negative times is wrapped around and stored at the extreme right end of the array r_k.

discrete convolution with a response function *of finite duration* N is a member of the discrete Fourier transform pair,

$$\sum_{k=-N/2+1}^{N/2} s_{j-k}\, r_k \quad \Longleftrightarrow \quad S_n R_n \tag{13.1.2}$$

Here $S_n,\ (n = 0, \ldots, N-1)$ is the discrete Fourier transform of the values $s_j,\ (j = 0, \ldots, N-1)$, while $R_n,\ (n = 0, \ldots, N-1)$ is the discrete Fourier transform of the values $r_k,\ (k = 0, \ldots, N-1)$. These values of r_k are the same ones as for the range $k = -N/2+1, \ldots, N/2$, but in wrap-around order, exactly as was described at the end of §12.2.

Treatment of End Effects by Zero Padding

The discrete convolution theorem presumes a set of two circumstances that are not universal. First, it assumes that the input signal is periodic, whereas real data often either go forever without repetition or else consist of one nonperiodic stretch of finite length. Second, the convolution theorem takes the duration of the response to be the same as the period of the data; they are both N. We need to work around these two constraints.

The second is very straightforward. Almost always, one is interested in a response function whose duration M is much shorter than the length of the data set N. In this case, you simply extend the response function to length N by padding it with zeros, i.e., define $r_k = 0$ for $M/2 \le k \le N/2$ and also for $-N/2+1 \le k \le -M/2+1$. Dealing with the first constraint is more challenging. Since the convolution theorem rashly assumes that the data are periodic, it will falsely "pollute" the first output channel $(r * s)_0$ with some wrapped-around data from the far end of the data stream s_{N-1}, s_{N-2}, etc. (See Figure 13.1.3.) So, we need to set up a buffer zone of zero-padded values at the end of the s_j vector, in order to make this pollution zero. How many zero values do we need in this buffer? Exactly as many as the most negative index for which the response function is nonzero. For example, if r_{-3} is nonzero, while $r_{-4}, r_{-5}, \ldots$ are all zero, then we need three zero pads at the end of the data: $s_{N-3} = s_{N-2} = s_{N-1} = 0$. These zeros will protect the first output channel $(r * s)_0$ from wrap-around pollution. It should be obvious that the second output channel $(r * s)_1$ and subsequent ones will also be protected by these same zeros. Let K denote the number of padding zeros, so that the last actual input data point is s_{N-K-1}.

What now about pollution of the very *last* output channel? Since the data now end with s_{N-K-1}, the last output channel of interest is $(r * s)_{N-K-1}$. This channel can be polluted by wrap-around from input channel s_0 unless the number K is also large enough to take care of the most positive index k for which the response function r_k is nonzero. For example, if r_0 through r_6 are nonzero, while $r_7, r_8 \ldots$ are all zero, then we need at least $K = 6$ padding zeros at the end of the data: $s_{N-6} = \ldots = s_{N-1} = 0$.

To summarize — we need to pad the data with a number of zeros *on one end* equal to the maximum positive duration *or* maximum negative duration of the response function, *whichever is larger*. (For a symmetric response function of duration M, you will need only $M/2$ zero pads.) Combining this operation with the

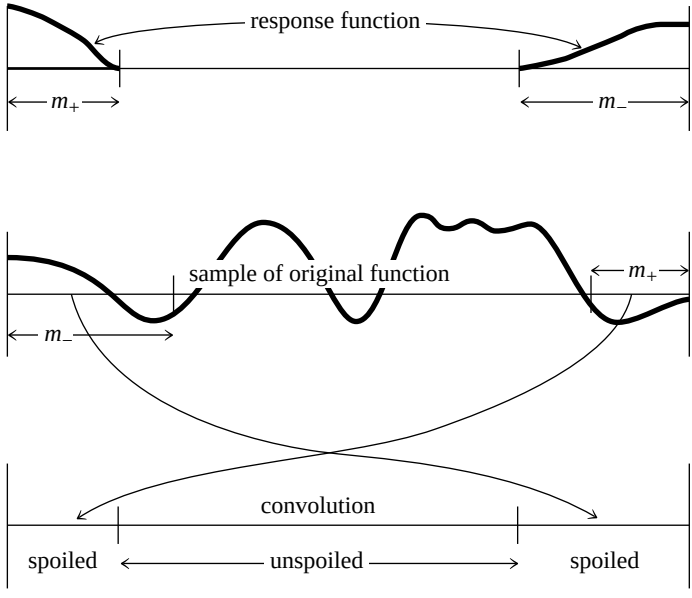


Figure 13.1.3. The wrap-around problem in convolving finite segments of a function. Not only must the response function wrap be viewed as cyclic, but so must the sampled original function. Therefore a portion at each end of the original function is erroneously wrapped around by convolution with the response function.

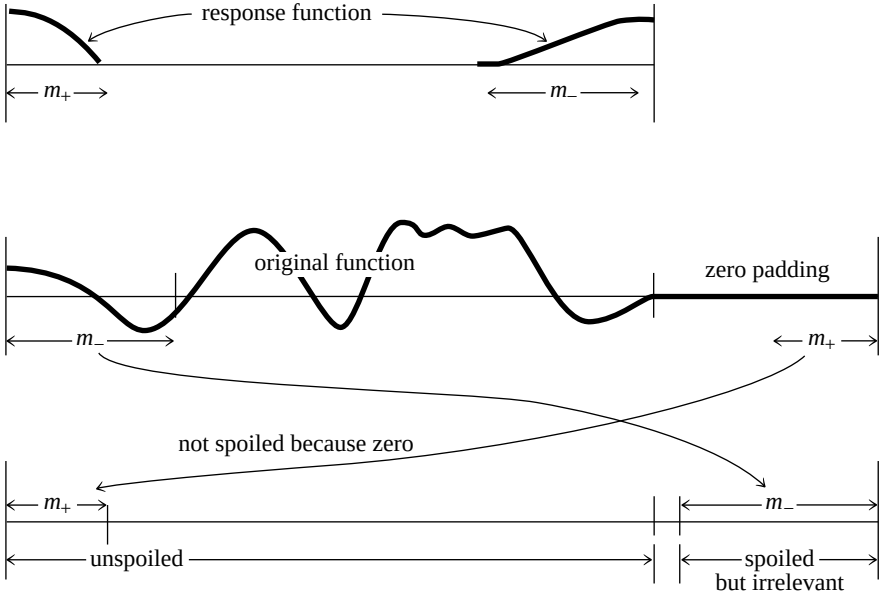


Figure 13.1.4. Zero padding as solution to the wrap-around problem. The original function is extended by zeros, serving a dual purpose: When the zeros wrap around, they do not disturb the true convolution; and while the original function wraps around onto the zero region, that region can be discarded.

padding of the response r_k described above, we effectively insulate the data from artifacts of undesired periodicity. Figure 13.1.4 illustrates matters.

Use of FFT for Convolution

The data, complete with zero padding, are now a set of real numbers s_j, $j = 0, \ldots, N-1$, and the response function is zero padded out to duration N and arranged in wrap-around order. (Generally this means that a large contiguous section of the r_k's, in the middle of that array, is zero, with nonzero values clustered at the two extreme ends of the array.) You now compute the discrete convolution as follows: Use the FFT algorithm to compute the discrete Fourier transform of s and of r. Multiply the two transforms together component by component, remembering that the transforms consist of complex numbers. Then use the FFT algorithm to take the inverse discrete Fourier transform of the products. The answer is the convolution $r * s$.

What about *deconvolution*? Deconvolution is the process of *undoing* the smearing in a data set that has occurred under the influence of a known response function, for example, because of the known effect of a less-than-perfect measuring apparatus. The defining equation of deconvolution is the same as that for convolution, namely (13.1.1), except now the left-hand side is taken to be known, and (13.1.1) is to be considered as a set of N linear equations for the unknown quantities s_j. Solving these simultaneous linear equations in the time domain of (13.1.1) is unrealistic in most cases, but the FFT renders the problem almost trivial. Instead of multiplying the transform of the signal and response to get the transform of the convolution, we just divide the transform of the (known) convolution by the transform of the response to get the transform of the deconvolved signal.

This procedure can go wrong *mathematically* if the transform of the response function is exactly zero for some value R_n, so that we can't divide by it. This indicates that the original convolution has truly lost all information at that one frequency, so that a reconstruction of that frequency component is not possible. You should be aware, however, that apart from mathematical problems, the process of deconvolution has other practical shortcomings. The process is generally quite sensitive to noise in the input data, and to the accuracy to which the response function r_k is known. Perfectly reasonable attempts at deconvolution can sometimes produce nonsense for these reasons. In such cases you may want to make use of the additional process of *optimal filtering*, which is discussed in §13.3.

Here is our routine for convolution and deconvolution, using the FFT as implemented in `four1` of §12.2. Since the data and response functions are real, not complex, both of their transforms can be taken simultaneously using `twofft`. Note, however, that two calls to `realft` should be substituted if `data` and `respns` have very different magnitudes, to minimize roundoff. The data are assumed to be stored in a real array `data` of length `n`, which must be an integer power of two. The response function is assumed to be stored in wrap-around order in a real array `respns` of length `m`. The value of `m` can be any *odd* integer less than or equal to `n`, since the first thing the program does is to recopy the response function into the appropriate wrap-around order in an array of length `n`. The answer is returned in `ans`, which is also used as working space.

```
      SUBROUTINE convlv(data,n,respns,m,isign,ans)
      INTEGER isign,m,n,NMAX
      REAL data(n),respns(n)
      COMPLEX ans(n)
      PARAMETER (NMAX=4096)                          Maximum anticipated size of FFT.
C     USES realft,twofft
```

Convolves or deconvolves a real data set `data(1:n)` (including any user-supplied zero padding) with a response function `respns`, stored in wrap-around order in a real array of length $m \leq n$. (`m` should be an odd integer.) Wrap-around order means that the first half of the array `respns` contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element `respns(m)`. On input `isign` is $+1$ for convolution, -1 for deconvolution. The answer is returned in the first `n` components of `ans`. However, `ans` must be supplied in the calling program with length at least `2*n`, for consistency with `twofft`. `n` MUST be an integer power of two.

```
      INTEGER i,no2
      COMPLEX fft(NMAX)
      do 11 i=1,(m-1)/2                              Put respns in array of length n.
          respns(n+1-i)=respns(m+1-i)
      enddo 11
      do 12 i=(m+3)/2,n-(m-1)/2                      Pad with zeros.
          respns(i)=0.0
      enddo 12
      call twofft(data,respns,fft,ans,n)             FFT both at once.
      no2=n/2
      do 13 i=1,no2+1
          if (isign.eq.1) then
              ans(i)=fft(i)*ans(i)/no2               Multiply FFTs to convolve.
          else if (isign.eq.-1) then
              if (abs(ans(i)).eq.0.0) pause 'deconvolving at response zero in convlv'
              ans(i)=fft(i)/ans(i)/no2               Divide FFTs to deconvolve.
          else
              pause 'no meaning for isign in convlv'
          endif
      enddo 13
      ans(1)=cmplx(real(ans(1)),real(ans(no2+1)))    Pack last element with first for realft.
      call realft(ans,n,-1)                          Inverse transform back to time domain.
      return
      END
```

Convolving or Deconvolving Very Large Data Sets

If your data set is so long that you do not want to fit it into memory all at once, then you must break it up into sections and convolve each section separately. Now, however, the treatment of end effects is a bit different. You have to worry not only about spurious wrap-around effects, but also about the fact that the ends of each section of data *should* have been influenced by data at the nearby ends of the immediately preceding and following sections of data, but were not so influenced since only one section of data is in the machine at a time.

There are two, related, standard solutions to this problem. Both are fairly obvious, so with a few words of description here, you ought to be able to implement them for yourself. The first solution is called the *overlap-save method*. In this technique you pad only the very beginning of the data with enough zeros to avoid wrap-around pollution. After this initial padding, you forget about zero padding altogether. Bring in a section of data and convolve or deconvolve it. Then throw out the points at each end that are polluted by wrap-around end effects. Output only the remaining good points in the middle. Now bring in the next section of data, but not all new data. The first points in each new section overlap the last points from

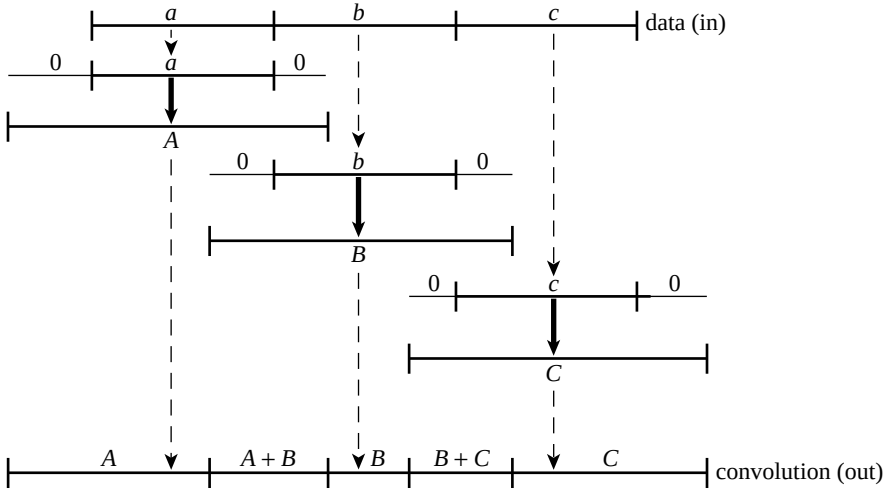


Figure 13.1.5. The overlap-add method for convolving a response with a very long signal. The signal data is broken up into smaller pieces. Each is zero padded at both ends and convolved (denoted by bold arrows in the figure). Finally the pieces are added back together, including the overlapping regions formed by the zero pads.

the preceding section of data. The sections must be overlapped sufficiently so that the polluted output points at the end of one section are recomputed as the first of the unpolluted output points from the subsequent section. With a bit of thought you can easily determine how many points to overlap and save.

The second solution, called the *overlap-add method*, is illustrated in Figure 13.1.5. Here you *don't* overlap the input data. Each section of data is disjoint from the others and is used exactly once. However, you carefully zero-pad it at both ends so that there is no wrap-around ambiguity in the output convolution or deconvolution. Now you overlap *and add* these sections of output. Thus, an output point near the end of one section will have the response due to the input points at the beginning of the next section of data properly added in to it, and likewise for an output point near the beginning of a section, *mutatis mutandis*.

Even when computer memory is available, there is some slight gain in computing speed in segmenting a long data set, since the FFTs' $N \log_2 N$ is slightly slower than linear in N. However, the log term is so slowly varying that you will often be much happier to avoid the bookkeeping complexities of the overlap-add or overlap-save methods: If it is practical to do so, just cram the whole data set into memory and FFT away. Then you will have more time for the finer things in life, some of which are described in succeeding sections of this chapter.

CITED REFERENCES AND FURTHER READING:

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.

13.2 Correlation and Autocorrelation Using the FFT

Correlation is the close mathematical cousin of convolution. It is in some ways simpler, however, because the two functions that go into a correlation are not as conceptually distinct as were the data and response functions that entered into convolution. Rather, in correlation, the functions are represented by different, but generally similar, data sets. We investigate their "correlation," by comparing them both directly superposed, and with one of them shifted left or right.

We have already defined in equation (12.0.10) the correlation between two continuous functions $g(t)$ and $h(t)$, which is denoted $\text{Corr}(g, h)$, and is a function of *lag* t. We will occasionally show this time dependence explicitly, with the rather awkward notation $\text{Corr}(g, h)(t)$. The correlation will be large at some value of t if the first function (g) is a close copy of the second (h) but lags it in time by t, i.e., if the first function is shifted to the right of the second. Likewise, the correlation will be large for some negative value of t if the first function *leads* the second, i.e., is shifted to the left of the second. The relation that holds when the two functions are interchanged is

$$\text{Corr}(g, h)(t) = \text{Corr}(h, g)(-t) \tag{13.2.1}$$

The discrete correlation of two sampled functions g_k and h_k, each periodic with period N, is defined by

$$\text{Corr}(g, h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \tag{13.2.2}$$

The *discrete correlation theorem* says that this discrete correlation of two real functions g and h is one member of the discrete Fourier transform pair

$$\text{Corr}(g, h)_j \Longleftrightarrow G_k H_k{}^* \tag{13.2.3}$$

where G_k and H_k are the discrete Fourier transforms of g_j and h_j, and the asterisk denotes complex conjugation. This theorem makes the same presumptions about the functions as those encountered for the discrete convolution theorem.

We can compute correlations using the FFT as follows: FFT the two data sets, multiply one resulting transform by the complex conjugate of the other, and inverse transform the product. The result (call it r_k) will formally be a complex vector of length N. However, it will turn out to have all its imaginary parts zero since the original data sets were both real. The components of r_k are the values of the correlation at different lags, with positive and negative lags stored in the by now familiar wrap-around order: The correlation at zero lag is in r_0, the first component; the correlation at lag 1 is in r_1, the second component; the correlation at lag -1 is in r_{N-1}, the last component; etc.

Just as in the case of convolution we have to consider end effects, since our data will not, in general, be periodic as intended by the correlation theorem. Here again, we can use zero padding. If you are interested in the correlation for lags as

13.2 Correlation and Autocorrelation Using the FFT

Correlation is the close mathematical cousin of convolution. It is in some ways simpler, however, because the two functions that go into a correlation are not as conceptually distinct as were the data and response functions that entered into convolution. Rather, in correlation, the functions are represented by different, but generally similar, data sets. We investigate their "correlation," by comparing them both directly superposed, and with one of them shifted left or right.

We have already defined in equation (12.0.10) the correlation between two continuous functions $g(t)$ and $h(t)$, which is denoted $\text{Corr}(g, h)$, and is a function of *lag* t. We will occasionally show this time dependence explicitly, with the rather awkward notation $\text{Corr}(g, h)(t)$. The correlation will be large at some value of t if the first function (g) is a close copy of the second (h) but lags it in time by t, i.e., if the first function is shifted to the right of the second. Likewise, the correlation will be large for some negative value of t if the first function *leads* the second, i.e., is shifted to the left of the second. The relation that holds when the two functions are interchanged is

$$\text{Corr}(g, h)(t) = \text{Corr}(h, g)(-t) \tag{13.2.1}$$

The discrete correlation of two sampled functions g_k and h_k, each periodic with period N, is defined by

$$\text{Corr}(g, h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \tag{13.2.2}$$

The *discrete correlation theorem* says that this discrete correlation of two real functions g and h is one member of the discrete Fourier transform pair

$$\text{Corr}(g, h)_j \Longleftrightarrow G_k H_k{}^* \tag{13.2.3}$$

where G_k and H_k are the discrete Fourier transforms of g_j and h_j, and the asterisk denotes complex conjugation. This theorem makes the same presumptions about the functions as those encountered for the discrete convolution theorem.

We can compute correlations using the FFT as follows: FFT the two data sets, multiply one resulting transform by the complex conjugate of the other, and inverse transform the product. The result (call it r_k) will formally be a complex vector of length N. However, it will turn out to have all its imaginary parts zero since the original data sets were both real. The components of r_k are the values of the correlation at different lags, with positive and negative lags stored in the by now familiar wrap-around order: The correlation at zero lag is in r_0, the first component; the correlation at lag 1 is in r_1, the second component; the correlation at lag -1 is in r_{N-1}, the last component; etc.

Just as in the case of convolution we have to consider end effects, since our data will not, in general, be periodic as intended by the correlation theorem. Here again, we can use zero padding. If you are interested in the correlation for lags as

large as $\pm K$, then you must append a buffer zone of K zeros at the end of both input data sets. If you want all possible lags from N data points (not a usual thing), then you will need to pad the data with an equal number of zeros; this is the extreme case. So here is the program:

```
      SUBROUTINE correl(data1,data2,n,ans)
      INTEGER n,NMAX
      REAL data1(n),data2(n)
      COMPLEX ans(n)
      PARAMETER (NMAX=4096)                    Maximum anticipated FFT size.
C     USES realft,twofft
        Computes the correlation of two real data sets data1(1:n) and data2(1:n) (includ-
        ing any user-supplied zero padding). n MUST be an integer power of two. The answer
        is returned as the first n points in ans stored in wrap-around order, i.e., correlations at
        increasingly negative lags are in ans(n) on down to ans(n/2+1), while correlations at
        increasingly positive lags are in ans(1) (zero lag) on up to ans(n/2). Note that ans
        must be supplied in the calling program with length at least 2*n, since it is also used as
        working space. Sign convention of this routine: if data1 lags data2, i.e., is shifted to the
        right of it, then ans will show a peak at positive lags.
      INTEGER i,no2
      COMPLEX fft(NMAX)
      call twofft(data1,data2,fft,ans,n)       Transform both data vectors at once.
      no2=n/2                                  Normalization for inverse FFT.
      do 11 i=1,no2+1
          ans(i)=fft(i)*conjg(ans(i))/float(no2)      Multiply to find FFT of their corre-
      enddo 11                                          lation.
      ans(1)=cmplx(real(ans(1)),real(ans(no2+1)))     Pack first and last into one element.
      call realft(ans,n,-1)                    Inverse transform gives correlation.
      return
      END
```

As in `convlv`, it would be better to substitute two calls to `realft` for the one call to `twofft`, if `data1` and `data2` have very different magnitudes, to minimize roundoff error.

The *discrete autocorrelation* of a sampled function g_j is just the discrete correlation of the function with itself. Obviously this is always symmetric with respect to positive and negative lags. Feel free to use the above routine `correl` to obtain autocorrelations, simply calling it with the same `data` vector in both arguments. If the inefficiency bothers you, routine `realft` can, of course, be used to transform the `data` vector instead.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §13–2.

13.3 Optimal (Wiener) Filtering with the FFT

There are a number of other tasks in numerical processing that are routinely handled with Fourier techniques. One of these is filtering for the removal of noise from a "corrupted" signal. The particular situation we consider is this: There is some underlying, uncorrupted signal $u(t)$ that we want to measure. The measurement process is imperfect, however, and what comes out of our measurement device is a corrupted signal $c(t)$. The signal $c(t)$ may be less than perfect in either or both of two respects. First, the apparatus may not have a perfect "delta-function" response,

large as $\pm K$, then you must append a buffer zone of K zeros at the end of both input data sets. If you want all possible lags from N data points (not a usual thing), then you will need to pad the data with an equal number of zeros; this is the extreme case. So here is the program:

```
      SUBROUTINE correl(data1,data2,n,ans)
      INTEGER n,NMAX
      REAL data1(n),data2(n)
      COMPLEX ans(n)
      PARAMETER (NMAX=4096)                    Maximum anticipated FFT size.
C     USES realft,twofft
        Computes the correlation of two real data sets data1(1:n) and data2(1:n) (includ-
        ing any user-supplied zero padding). n MUST be an integer power of two. The answer
        is returned as the first n points in ans stored in wrap-around order, i.e., correlations at
        increasingly negative lags are in ans(n) on down to ans(n/2+1), while correlations at
        increasingly positive lags are in ans(1) (zero lag) on up to ans(n/2). Note that ans
        must be supplied in the calling program with length at least 2*n, since it is also used as
        working space. Sign convention of this routine: if data1 lags data2, i.e., is shifted to the
        right of it, then ans will show a peak at positive lags.
      INTEGER i,no2
      COMPLEX fft(NMAX)
      call twofft(data1,data2,fft,ans,n)       Transform both data vectors at once.
      no2=n/2                                  Normalization for inverse FFT.
      do 11 i=1,no2+1
          ans(i)=fft(i)*conjg(ans(i))/float(no2)      Multiply to find FFT of their corre-
      enddo 11                                          lation.
      ans(1)=cmplx(real(ans(1)),real(ans(no2+1)))     Pack first and last into one element.
      call realft(ans,n,-1)                    Inverse transform gives correlation.
      return
      END
```

As in `convlv`, it would be better to substitute two calls to `realft` for the one call to `twofft`, if `data1` and `data2` have very different magnitudes, to minimize roundoff error.

The *discrete autocorrelation* of a sampled function g_j is just the discrete correlation of the function with itself. Obviously this is always symmetric with respect to positive and negative lags. Feel free to use the above routine `correl` to obtain autocorrelations, simply calling it with the same `data` vector in both arguments. If the inefficiency bothers you, routine `realft` can, of course, be used to transform the `data` vector instead.

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §13–2.

13.3 Optimal (Wiener) Filtering with the FFT

There are a number of other tasks in numerical processing that are routinely handled with Fourier techniques. One of these is filtering for the removal of noise from a "corrupted" signal. The particular situation we consider is this: There is some underlying, uncorrupted signal $u(t)$ that we want to measure. The measurement process is imperfect, however, and what comes out of our measurement device is a corrupted signal $c(t)$. The signal $c(t)$ may be less than perfect in either or both of two respects. First, the apparatus may not have a perfect "delta-function" response,

so that the true signal $u(t)$ is convolved with (smeared out by) some known response function $r(t)$ to give a smeared signal $s(t)$,

$$s(t) = \int_{-\infty}^{\infty} r(t-\tau)u(\tau)\, d\tau \quad \text{or} \quad S(f) = R(f)U(f) \tag{13.3.1}$$

where S, R, U are the Fourier transforms of s, r, u, respectively. Second, the measured signal $c(t)$ may contain an additional component of noise $n(t)$,

$$c(t) = s(t) + n(t) \tag{13.3.2}$$

We already know how to deconvolve the effects of the response function r in the absence of any noise (§13.1); we just divide $C(f)$ by $R(f)$ to get a deconvolved signal. We now want to treat the analogous problem when noise is present. Our task is to find the *optimal filter*, $\phi(t)$ or $\Phi(f)$, which, when applied to the measured signal $c(t)$ or $C(f)$, and then deconvolved by $r(t)$ or $R(f)$, produces a signal $\widetilde{u}(t)$ or $\widetilde{U}(f)$ that is as close as possible to the uncorrupted signal $u(t)$ or $U(f)$. In other words we will estimate the true signal U by

$$\widetilde{U}(f) = \frac{C(f)\Phi(f)}{R(f)} \tag{13.3.3}$$

In what sense is $\widetilde{U}$ to be close to U? We ask that they be *close in the least-square sense*

$$\int_{-\infty}^{\infty} |\widetilde{u}(t) - u(t)|^2 \, dt = \int_{-\infty}^{\infty} \left|\widetilde{U}(f) - U(f)\right|^2 \, df \quad \text{is minimized.} \tag{13.3.4}$$

Substituting equations (13.3.3) and (13.3.2), the right-hand side of (13.3.4) becomes

$$\begin{aligned} &\int_{-\infty}^{\infty} \left| \frac{[S(f)+N(f)]\Phi(f)}{R(f)} - \frac{S(f)}{R(f)} \right|^2 df \\ &= \int_{-\infty}^{\infty} |R(f)|^{-2} \left\{ |S(f)|^2 \, |1-\Phi(f)|^2 + |N(f)|^2 \, |\Phi(f)|^2 \right\} df \end{aligned} \tag{13.3.5}$$

The signal S and the noise N are *uncorrelated*, so their cross product, when integrated over frequency f, gave zero. (This is practically the *definition* of what we mean by noise!) Obviously (13.3.5) will be a minimum if and only if the integrand is minimized with respect to $\Phi(f)$ at every value of f. Let us search for such a solution where $\Phi(f)$ is a real function. Differentiating with respect to Φ, and setting the result equal to zero gives

$$\Phi(f) = \frac{|S(f)|^2}{|S(f)|^2 + |N(f)|^2} \tag{13.3.6}$$

This is the formula for the optimal filter $\Phi(f)$.

Notice that equation (13.3.6) involves S, the smeared signal, and N, the noise. The two of these add up to be C, the measured signal. Equation (13.3.6) does not

contain U, the "true" signal. This makes for an important simplification: The optimal filter can be determined independently of the determination of the deconvolution function that relates S and U.

To determine the optimal filter from equation (13.3.6) we need some way of separately estimating $|S|^2$ and $|N|^2$. There is no way to do this from the measured signal C alone without some other information, or some assumption or guess. Luckily, the extra information is often easy to obtain. For example, we can sample a long stretch of data $c(t)$ and plot its power spectral density using equations (12.0.14), (12.1.8), and (12.1.5). This quantity is proportional to the sum $|S|^2 + |N|^2$, so we have

$$|S(f)|^2 + |N(f)|^2 \approx P_c(f) = |C(f)|^2 \qquad 0 \le f < f_c \tag{13.3.7}$$

(More sophisticated methods of estimating the power spectral density will be discussed in §13.4 and §13.7, but the estimation above is almost always good enough for the optimal filter problem.) The resulting plot (see Figure 13.3.1) will often immediately show the spectral signature of a signal sticking up above a continuous noise spectrum. The noise spectrum may be flat, or tilted, or smoothly varying; it doesn't matter, as long as we can guess a reasonable hypothesis as to what it is. Draw a smooth curve through the noise spectrum, extrapolating it into the region dominated by the signal as well. Now draw a smooth curve through the signal plus noise power. The difference between these two curves is your smooth "model" of the signal power. The quotient of your model of signal power to your model of signal plus noise power is the optimal filter $\Phi(f)$. [Extend it to negative values of f by the formula $\Phi(-f) = \Phi(f)$.] Notice that $\Phi(f)$ will be close to unity where the noise is negligible, and close to zero where the noise is dominant. That is how it does its job! The intermediate dependence given by equation (13.3.6) just turns out to be the optimal way of going in between these two extremes.

Because the optimal filter results from a minimization problem, the quality of the results obtained by optimal filtering differs from the true optimum by an amount that is *second order* in the precision to which the optimal filter is determined. In other words, even a fairly crudely determined optimal filter (sloppy, say, at the 10 percent level) can give excellent results when it is applied to data. That is why the separation of the measured signal C into signal and noise components S and N can usefully be done "by eye" from a crude plot of power spectral density. All of this may give you thoughts about iterating the procedure we have just described. For example, after designing a filter with response $\Phi(f)$ and using it to make a respectable guess at the signal $\widetilde{U}(f) = \Phi(f)C(f)/R(f)$, you might turn about and regard $\widetilde{U}(f)$ as a fresh new signal which you could improve even further with the same filtering technique. Don't waste your time on this line of thought. The scheme converges to a signal of $S(f) = 0$. Converging iterative methods do exist; this just isn't one of them.

You can use the routine `four1` (§12.2) or `realft` (§12.3) to FFT your data when you are constructing an optimal filter. To apply the filter to your data, you can use the methods described in §13.1. The specific routine `convlv` is not needed for optimal filtering, since your filter is constructed in the frequency domain to begin with. If you are also deconvolving your data with a known response function, however, you can modify `convlv` to multiply by your optimal filter just before it takes the inverse Fourier transform.

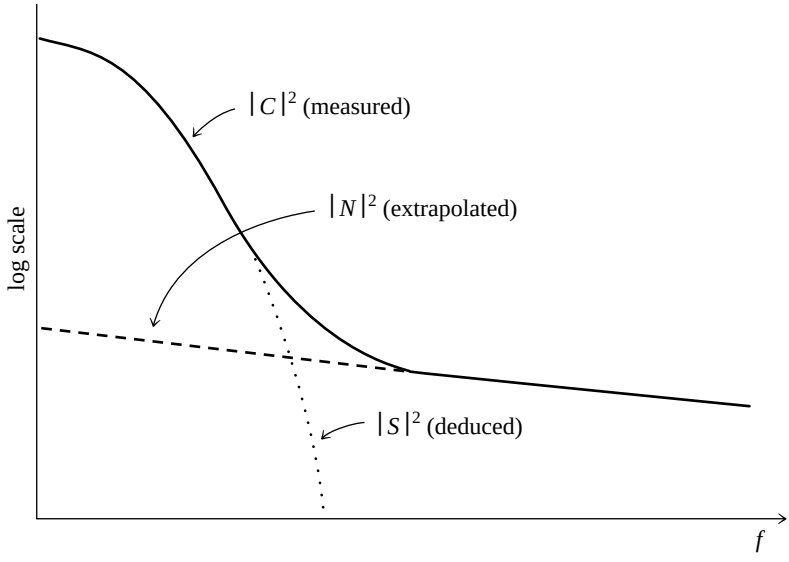


Figure 13.3.1. Optimal (Wiener) filtering. The power spectrum of signal plus noise shows a signal peak added to a noise tail. The tail is extrapolated back into the signal region as a "noise model." Subtracting gives the "signal model." The models need not be accurate for the method to be useful. A simple algebraic combination of the models gives the optimal filter (see text).

CITED REFERENCES AND FURTHER READING:

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

13.4 Power Spectrum Estimation Using the FFT

In the previous section we "informally" estimated the power spectral density of a function $c(t)$ by taking the modulus-squared of the discrete Fourier transform of some finite, sampled stretch of it. In this section we'll do roughly the same thing, but with considerably greater attention to details. Our attention will uncover some surprises.

The first detail is power spectrum (also called a power spectral density or PSD) normalization. In general there is *some* relation of proportionality between a measure of the squared amplitude of the function and a measure of the amplitude of the PSD. Unfortunately there are several different conventions for describing the normalization in each domain, and many opportunities for getting wrong the relationship between the two domains. Suppose that our function $c(t)$ is sampled at N points to produce values $c_0 \dots c_{N-1}$, and that these points span a range of time T, that is $T = (N-1)\Delta$, where Δ is the sampling interval. Then here are several

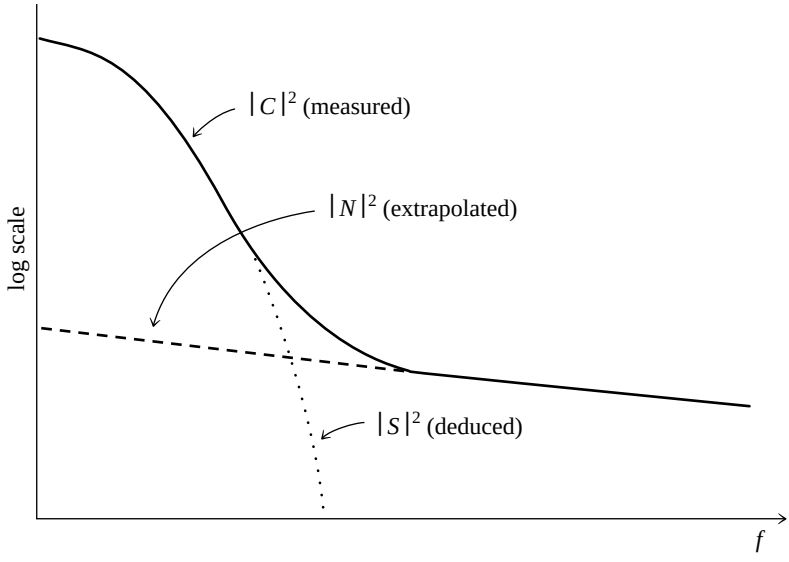


Figure 13.3.1. Optimal (Wiener) filtering. The power spectrum of signal plus noise shows a signal peak added to a noise tail. The tail is extrapolated back into the signal region as a "noise model." Subtracting gives the "signal model." The models need not be accurate for the method to be useful. A simple algebraic combination of the models gives the optimal filter (see text).

CITED REFERENCES AND FURTHER READING:

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

13.4 Power Spectrum Estimation Using the FFT

In the previous section we "informally" estimated the power spectral density of a function $c(t)$ by taking the modulus-squared of the discrete Fourier transform of some finite, sampled stretch of it. In this section we'll do roughly the same thing, but with considerably greater attention to details. Our attention will uncover some surprises.

The first detail is power spectrum (also called a power spectral density or PSD) normalization. In general there is *some* relation of proportionality between a measure of the squared amplitude of the function and a measure of the amplitude of the PSD. Unfortunately there are several different conventions for describing the normalization in each domain, and many opportunities for getting wrong the relationship between the two domains. Suppose that our function $c(t)$ is sampled at N points to produce values $c_0 \dots c_{N-1}$, and that these points span a range of time T, that is $T = (N-1)\Delta$, where Δ is the sampling interval. Then here are several

different descriptions of the total power:

$$\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“sum squared amplitude”} \tag{13.4.1}$$

$$\frac{1}{T}\int_0^T |c(t)|^2 \, dt \approx \frac{1}{N}\sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“mean squared amplitude”} \tag{13.4.2}$$

$$\int_0^T |c(t)|^2 \, dt \approx \Delta \sum_{j=0}^{N-1} |c_j|^2 \equiv \text{“time-integral squared amplitude”} \tag{13.4.3}$$

PSD estimators, as we shall see, have an even greater variety. In this section, we consider a class of them that give estimates at discrete values of frequency f_i, where i will range over integer values. In the next section, we will learn about a different class of estimators that produce estimates that are continuous functions of frequency f. Even if it is agreed always to relate the PSD normalization to a particular description of the function normalization (e.g., 13.4.2), there are at least the following possibilities: The PSD is

- defined for discrete positive, zero, and negative frequencies, and its sum over these is the function mean squared amplitude
- defined for zero and discrete positive frequencies only, and its sum over these is the function mean squared amplitude
- defined in the Nyquist interval from $-f_c$ to f_c, and its integral over this range is the function mean squared amplitude
- defined from 0 to f_c, and its integral over this range is the function mean squared amplitude

It *never* makes sense to integrate the PSD of a sampled function outside of the Nyquist interval $-f_c$ and f_c since, according to the sampling theorem, power there will have been aliased into the Nyquist interval.

It is hopeless to define enough notation to distinguish all possible combinations of normalizations. In what follows, we use the notation $P(f)$ to mean *any* of the above PSDs, stating in each instance how the particular $P(f)$ is normalized. Beware the inconsistent notation in the literature.

The method of power spectrum estimation used in the previous section is a simple version of an estimator called, historically, the *periodogram*. If we take an N-point sample of the function $c(t)$ at equal intervals and use the FFT to compute its discrete Fourier transform

$$C_k = \sum_{j=0}^{N-1} c_j \, e^{2\pi i j k/N} \qquad k = 0, \ldots, N-1 \tag{13.4.4}$$

then the periodogram estimate of the power spectrum is defined at $N/2 + 1$

frequencies as

$$P(0) = P(f_0) = \frac{1}{N^2}\left|C_0\right|^2$$
$$P(f_k) = \frac{1}{N^2}\left[\left|C_k\right|^2 + \left|C_{N-k}\right|^2\right] \qquad k = 1, 2, \ldots, \left(\frac{N}{2} - 1\right) \tag{13.4.5}$$
$$P(f_c) = P(f_{N/2}) = \frac{1}{N^2}\left|C_{N/2}\right|^2$$

where f_k is defined only for the zero and positive frequencies

$$f_k \equiv \frac{k}{N\Delta} = 2f_c\frac{k}{N} \qquad k = 0, 1, \ldots, \frac{N}{2} \tag{13.4.6}$$

By Parseval's theorem, equation (12.1.10), we see immediately that equation (13.4.5) is normalized so that the sum of the $N/2 + 1$ values of P is equal to the mean squared amplitude of the function c_j.

We must now ask this question. In what sense is the periodogram estimate (13.4.5) a "true" estimator of the power spectrum of the underlying function $c(t)$? You can find the answer treated in considerable detail in the literature cited (see, e.g., [1] for an introduction). Here is a summary.

First, is the *expectation value* of the periodogram estimate equal to the power spectrum, i.e., is the estimator correct on average? Well, yes and no. We wouldn't really expect one of the $P(f_k)$'s to equal the continuous $P(f)$ at *exactly* f_k, since f_k is supposed to be representative of a whole frequency "bin" extending from halfway from the preceding discrete frequency to halfway to the next one. We *should* be expecting the $P(f_k)$ to be some kind of average of $P(f)$ over a narrow window function centered on its f_k. For the periodogram estimate (13.4.6) that window function, as a function of s the frequency offset *in bins*, is

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 \tag{13.4.7}$$

Notice that $W(s)$ has oscillatory lobes but, apart from these, falls off only about as $W(s) \approx (\pi s)^{-2}$. This is not a very rapid fall-off, and it results in significant *leakage* (that is the technical term) from one frequency to another in the periodogram estimate. Notice also that $W(s)$ happens to be zero for s equal to a nonzero integer. This means that if the function $c(t)$ is a pure sine wave of frequency exactly equal to one of the f_k's, then there will be *no* leakage to adjacent f_k's. But this is not the characteristic case! If the frequency is, say, one-third of the way between two adjacent f_k's, then the leakage will extend *well* beyond those two adjacent bins. The solution to the problem of leakage is called *data windowing*, and we will discuss it below.

Turn now to another question about the periodogram estimate. What is the variance of that estimate as N goes to infinity? In other words, as we take more sampled points from the original function (either sampling a longer stretch of data at the same sampling rate, or else by resampling the same stretch of data with a faster sampling rate), then how much more accurate do the estimates P_k become? The unpleasant answer is that the periodogram estimates *do not become more accurate at all!* In fact, the variance of the periodogram estimate at a frequency f_k is always

equal to the square of its expectation value at that frequency. In other words, the standard deviation is always 100 percent of the value, independent of N! How can this be? Where did all the information go as we added points? It all went into producing estimates at a greater number of discrete frequencies f_k. If we sample a longer run of data using the same sampling rate, then the Nyquist critical frequency f_c is unchanged, but we now have finer frequency resolution (more f_k's) within the Nyquist frequency interval; alternatively, if we sample the same length of data with a finer sampling interval, then our frequency resolution is unchanged, but the Nyquist range now extends up to a higher frequency. In neither case do the additional samples reduce the variance of any one particular frequency's estimated PSD.

You don't have to live with PSD estimates with 100 percent standard deviations, however. You simply have to know some techniques for reducing the variance of the estimates. Here are two techniques that are very nearly identical mathematically, though different in implementation. The first is to compute a periodogram estimate with finer discrete frequency spacing than you really need, and then to sum the periodogram estimates at K consecutive discrete frequencies to get one "smoother" estimate at the mid frequency of those K. The variance of that summed estimate will be smaller than the estimate itself by a factor of exactly $1/K$, i.e., the standard deviation will be smaller than 100 percent by a factor $1/\sqrt{K}$. Thus, to estimate the power spectrum at $M+1$ discrete frequencies between 0 and f_c inclusive, you begin by taking the FFT of $2MK$ points (which number had better be an integer power of two!). You then take the modulus square of the resulting coefficients, add positive and negative frequency pairs, and divide by $(2MK)^2$, all according to equation (13.4.5) with $N = 2MK$. Finally, you "bin" the results into summed (not averaged) groups of K. This procedure is very easy to program, so we will not bother to give a routine for it. The reason that you sum, rather than average, K consecutive points is so that your final PSD estimate will preserve the normalization property that the sum of its $M+1$ values equals the mean square value of the function.

A second technique for estimating the PSD at $M+1$ discrete frequencies in the range 0 to f_c is to partition the original sampled data into K segments each of $2M$ consecutive sampled points. Each segment is separately FFT'd to produce a periodogram estimate (equation 13.4.5 with $N \equiv 2M$). Finally, the K periodogram estimates are averaged at each frequency. It is this final averaging that reduces the variance of the estimate by a factor K (standard deviation by $\sqrt{K}$). This second technique is computationally more efficient than the first technique above by a modest factor, since it is logarithmically more efficient to take many shorter FFTs than one longer one. The principal advantage of the second technique, however, is that only $2M$ data points are manipulated at a single time, not $2KM$ as in the first technique. This means that the second technique is the natural choice for processing long runs of data, as from a magnetic tape or other data record. We will give a routine later for implementing this second technique, but we need first to return to the matters of leakage and data windowing which were brought up after equation (13.4.7) above.

Data Windowing

The purpose of data windowing is to modify equation (13.4.7), which expresses the relation between the spectral estimate P_k at a discrete frequency and the actual underlying continuous spectrum $P(f)$ at nearby frequencies. In general, the spectral

power in one "bin" k contains leakage from frequency components that are actually s bins away, where s is the independent variable in equation (13.4.7). There is, as we pointed out, quite substantial leakage even from moderately large values of s.

When we select a run of N sampled points for periodogram spectral estimation, we are in effect multiplying an infinite run of sampled data c_j by a window function in time, one that is zero except during the total sampling time $N\Delta$, and is unity during that time. In other words, the data are windowed by a square window function. By the convolution theorem (12.0.9; but interchanging the roles of f and t), the Fourier transform of the product of the data with this square window function is equal to the convolution of the data's Fourier transform with the window's Fourier transform. In fact, we determined equation (13.4.7) as nothing more than the square of the discrete Fourier transform of the unity window function.

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 = \frac{1}{N^2}\left|\sum_{k=0}^{N-1} e^{2\pi i s k/N}\right|^2 \tag{13.4.8}$$

The reason for the leakage at large values of s, is that the square window function turns on and off so rapidly. Its Fourier transform has substantial components at high frequencies. To remedy this situation, we can multiply the input data $c_j,\ j = 0, \ldots, N-1$ by a window function w_j that changes more gradually from zero to a maximum and then back to zero as j ranges from 0 to N. In this case, the equations for the periodogram estimator (13.4.4–13.4.5) become

$$D_k \equiv \sum_{j=0}^{N-1} c_j w_j\, e^{2\pi i j k/N} \qquad k = 0, \ldots, N-1 \tag{13.4.9}$$

$$\begin{aligned} P(0) &= P(f_0) = \frac{1}{W_{ss}}\,|D_0|^2 \\ P(f_k) &= \frac{1}{W_{ss}}\left[|D_k|^2 + |D_{N-k}|^2\right] \qquad k = 1, 2, \ldots, \left(\frac{N}{2} - 1\right) \\ P(f_c) &= P(f_{N/2}) = \frac{1}{W_{ss}}\,|D_{N/2}|^2 \end{aligned} \tag{13.4.10}$$

where W_{ss} stands for "window squared and summed,"

$$W_{ss} \equiv N \sum_{j=0}^{N-1} w_j^2 \tag{13.4.11}$$

and f_k is given by (13.4.6). The more general form of (13.4.7) can now be written in terms of the window function w_j as

$$\begin{aligned} W(s) &= \frac{1}{W_{ss}}\left|\sum_{k=0}^{N-1} e^{2\pi i s k/N} w_k\right|^2 \\ &\approx \frac{1}{W_{ss}}\left|\int_{-N/2}^{N/2} \cos(2\pi s k/N) w(k - N/2)\, dk\right|^2 \end{aligned} \tag{13.4.12}$$

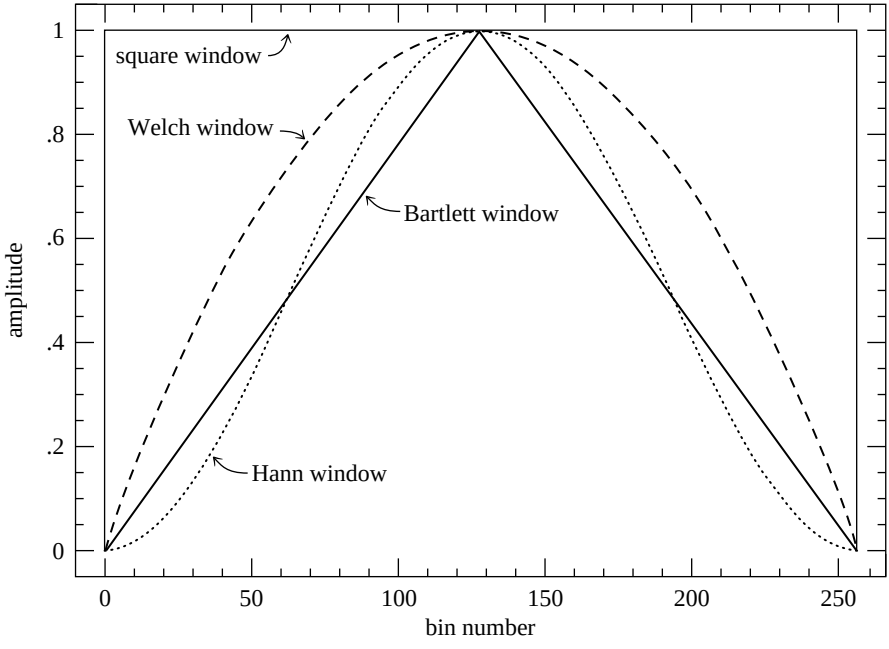


Figure 13.4.1. Window functions commonly used in FFT power spectral estimation. The data segment, here of length 256, is multiplied (bin by bin) by the window function before the FFT is computed. The square window, which is equivalent to no windowing, is least recommended. The Welch and Bartlett windows are good choices.

Here the approximate equality is useful for practical estimates, and holds for any window that is left-right symmetric (the usual case), and for $s \ll N$ (the case of interest for estimating leakage into nearby bins). The continuous function $w(k-N/2)$ in the integral is meant to be some smooth function that passes through the points w_k.

There is a lot of perhaps unnecessary lore about choice of a window function, and practically every function that rises from zero to a peak and then falls again has been named after someone. A few of the more common (also shown in Figure 13.4.1) are:

$$w_j = 1 - \left| \frac{j - \frac{1}{2}N}{\frac{1}{2}N} \right| \equiv \text{“Bartlett window”} \tag{13.4.13}$$

(The “Parzen window” is very similar to this.)

$$w_j = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi j}{N}\right)\right] \equiv \text{“Hann window”} \tag{13.4.14}$$

(The “Hamming window” is similar but does not go exactly to zero at the ends.)

$$w_j = 1 - \left(\frac{j - \frac{1}{2}N}{\frac{1}{2}N}\right)^2 \equiv \text{“Welch window”} \tag{13.4.15}$$

We are inclined to follow Welch in recommending that you use either (13.4.13) or (13.4.15) in practical work. However, at the level of this book, there is effectively

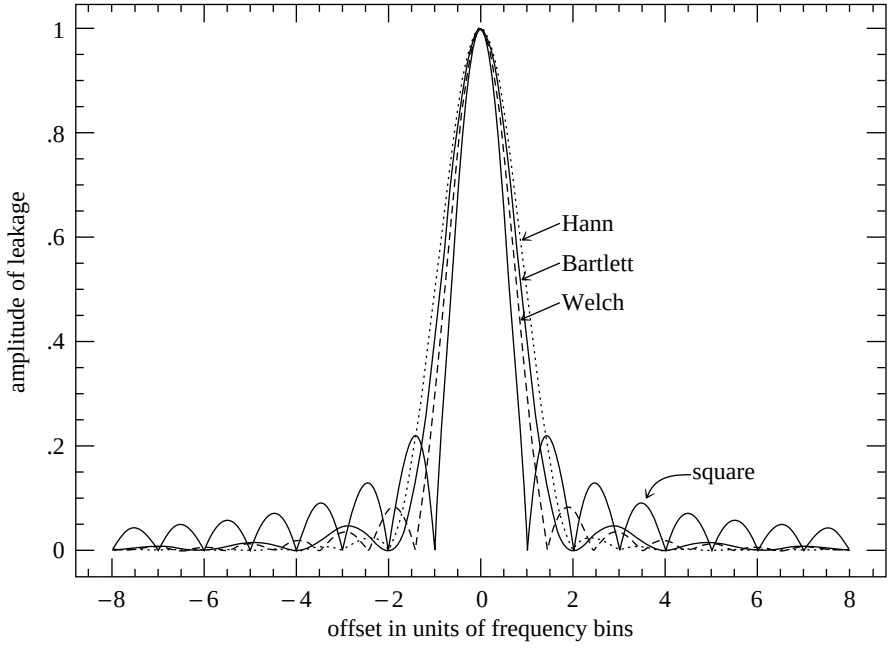


Figure 13.4.2. Leakage functions for the window functions of Figure 13.4.1. A signal whose frequency is actually located at zero offset "leaks" into neighboring bins with the amplitude shown. The purpose of windowing is to reduce the leakage at large offsets, where square (no) windowing has large sidelobes. Offset can have a fractional value, since the actual signal frequency can be located between two frequency bins of the FFT.

no difference between any of these (or similar) window functions. Their difference lies in subtle trade-offs among the various figures of merit that can be used to describe the narrowness or peakedness of the spectral leakage functions computed by (13.4.12). These figures of merit have such names as: *highest sidelobe level (dB), sidelobe fall-off (dB per octave), equivalent noise bandwidth (bins), 3-dB bandwidth (bins), scallop loss (dB), worst case process loss (dB)*. Roughly speaking, the principal trade-off is between making the central peak as narrow as possible versus making the tails of the distribution fall off as rapidly as possible. For details, see (e.g.) [2]. Figure 13.4.2 plots the leakage amplitudes for several windows already discussed.

There is particularly a lore about window functions that rise smoothly from zero to unity in the first small fraction (say 10 percent) of the data, then stay at unity until the last small fraction (again say 10 percent) of the data, during which the window function falls smoothly back to zero. These windows will squeeze a little bit of extra narrowness out of the main lobe of the leakage function (never as much as a factor of two, however), but trade this off by widening the leakage tail by a significant factor (e.g., the reciprocal of 10 percent, a factor of ten). If we distinguish between the *width* of a window (number of samples for which it is at its maximum value) and its *rise/fall time* (number of samples during which it rises and falls); and if we distinguish between the *FWHM* (full width to half maximum value) of the leakage function's main lobe and the *leakage width* (full width that contains half of the spectral power that is not contained in the main lobe); then

these quantities are related roughly by

$$(\text{FWHM in bins}) \approx \frac{N}{(\text{window width})} \tag{13.4.16}$$

$$(\text{leakage width in bins}) \approx \frac{N}{(\text{window rise/fall time})} \tag{13.4.17}$$

For the windows given above in (13.4.13)–(13.4.15), the effective window widths and the effective window rise/fall times are both of order $\frac{1}{2}N$. Generally speaking, we feel that the advantages of windows whose rise and fall times are only small fractions of the data length are minor or nonexistent, and we avoid using them. One sometimes hears it said that flat-topped windows "throw away less of the data," but we will now show you a better way of dealing with that problem by use of overlapping data segments.

Let us now suppose that we have chosen a window function, and that we are ready to segment the data into K segments of $N = 2M$ points. Each segment will be FFT'd, and the resulting K periodograms will be averaged together to obtain a PSD estimate at $M + 1$ frequency values from 0 to f_c. We must now distinguish between two possible situations. We might want to obtain the smallest variance from a fixed amount of computation, without regard to the number of data points used. This will generally be the goal when the data are being gathered in real time, with the data-reduction being computer-limited. Alternatively, we might want to obtain the smallest variance from a fixed number of available sampled data points. This will generally be the goal in cases where the data are already recorded and we are analyzing it after the fact.

In the first situation (smallest spectral variance per computer operation), it is best to segment the data without any overlapping. The first $2M$ data points constitute segment number 1; the next $2M$ data points constitute segment number 2; and so on, up to segment number K, for a total of $2KM$ sampled points. The variance in this case, relative to a single segment, is reduced by a factor K.

In the second situation (smallest spectral variance per data point), it turns out to be optimal, or very nearly optimal, to overlap the segments by one half of their length. The first and second sets of M points are segment number 1; the second and third sets of M points are segment number 2; and so on, up to segment number K, which is made of the Kth and $K + 1$st sets of M points. The total number of sampled points is therefore $(K+1)M$, just over half as many as with nonoverlapping segments. The reduction in the variance is not a full factor of K, since the segments are not statistically independent. It can be shown that the variance is instead reduced by a factor of about $9K/11$ (see the paper by Welch in [3]). This is, however, significantly better than the reduction of about $K/2$ that would have resulted if the same *number* of data points were segmented without overlapping.

We can now codify these ideas into a routine for spectral estimation. While we generally avoid input/output coding, we make an exception here to show how data are read sequentially in one pass through a data file (here `FORTRAN` Unit 9). Only a small fraction of the data is in memory at any one time. Note that `spctrm` returns the power at M, not $M + 1$, frequencies, omitting the component $P(f_c)$ at the Nyquist frequency. It would also be straightforward to include that component.

```
      SUBROUTINE spctrm(p,m,k,ovrlap,w1,w2)
      INTEGER k,m
      REAL p(m),w1(4*m),w2(m)
      LOGICAL ovrlap                  True for overlapping segments, false otherwise.
C     USES four1
        Reads data from input unit 9 and returns as p(j) the data's power (mean square amplitude)
        at frequency (j-1)/(2*m) cycles per gridpoint, for j=1,2,...,m, based on (2*k+1)*m
        data points (if ovrlap is set .true.) or 4*k*m data points (if ovrlap is set .false.).
        The number of segments of the data is 2*k in both cases: The routine calls four1 k
        times, each call with 2 partitions each of 2*m real data points. w1(1:4*m) and w2(1:m)
        are user-supplied workspaces.
      INTEGER j,j2,joff,joffn,kk,m4,m43,m44,mm
      REAL den,facm,facp,sumw,w,window
      window(j)=(1.-abs(((j-1)-facm)*facp))     Statement function defines Bartlett window.
C     window(j)=1.                              Alternative for square window.
C     window(j)=(1.-(((j-1)-facm)*facp)**2)     Alternative for Welch window.
      mm=m+m                                    Useful factors.
      m4=mm+mm
      m44=m4+4
      m43=m4+3
      den=0.
      facm=m                                    Factors used by the window statement function.
      facp=1./m
      sumw=0.                                   Accumulate the squared sum of the weights.
      do 11 j=1,mm
        sumw=sumw+window(j)**2
      enddo 11
      do 12 j=1,m                               Initialize the spectrum to zero.
        p(j)=0.
      enddo 12
      if(ovrlap)then                            Initialize the "save" half-buffer.
        read (9,*) (w2(j),j=1,m)
      endif
      do 18 kk=1,k                              Loop over data set segments in groups of two.
        do 15 joff=-1,0,1                       Get two complete segments into workspace.
          if (ovrlap) then
            do 13 j=1,m
              w1(joff+j+j)=w2(j)
            enddo 13
            read (9,*) (w2(j),j=1,m)
            joffn=joff+mm
            do 14 j=1,m
              w1(joffn+j+j)=w2(j)
            enddo 14
          else
            read (9,*) (w1(j),j=joff+2,m4,2)
          endif
        enddo 15
        do 16 j=1,mm                            Apply the window to the data.
          j2=j+j
          w=window(j)
          w1(j2)=w1(j2)*w
          w1(j2-1)=w1(j2-1)*w
        enddo 16
        call four1(w1,mm,1)                     Fourier transform the windowed data.
        p(1)=p(1)+w1(1)**2+w1(2)**2             Sum results into previous segments.
        do 17 j=2,m
          j2=j+j
          p(j)=p(j)+w1(j2)**2+w1(j2-1)**2
     *         +w1(m44-j2)**2+w1(m43-j2)**2
        enddo 17
        den=den+sumw
      enddo 18
      den=m4*den                                Correct normalization.
```

```
do 19 j=1,m
    p(j)=p(j)/den                    Normalize the output.
enddo 19
return
END
```

CITED REFERENCES AND FURTHER READING:

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Harris, F.J. 1978, *Proceedings of the IEEE*, vol. 66, pp. 51–83. [2]

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), paper by P.D. Welch. [3]

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

13.5 Digital Filtering in the Time Domain

Suppose that you have a signal that you want to filter digitally. For example, perhaps you want to apply *high-pass* or *low-pass* filtering, to eliminate noise at low or high frequencies respectively; or perhaps the interesting part of your signal lies only in a certain frequency band, so that you need a *bandpass* filter. Or, if your measurements are contaminated by 60 Hz power-line interference, you may need a *notch filter* to remove only a narrow band around that frequency. This section speaks particularly about the case in which you have chosen to do such filtering in the time domain.

Before continuing, we hope you will reconsider this choice. Remember how convenient it is to filter in the Fourier domain. You just take your whole data record, FFT it, multiply the FFT output by a filter function $\mathcal{H}(f)$, and then do an inverse FFT to get back a filtered data set in time domain. Here is some additional background on the Fourier technique that you will want to take into account.

- Remember that you must define your filter function $\mathcal{H}(f)$ for both positive and negative frequencies, and that the magnitude of the frequency extremes is always the Nyquist frequency $1/(2\Delta)$, where Δ is the sampling interval. The magnitude of the smallest nonzero frequencies in the FFT is $\pm 1/(N\Delta)$, where N is the number of (complex) points in the FFT. The positive and negative frequencies to which this filter are applied are arranged in wrap-around order.
- If the measured data are real, and you want the filtered output also to be real, then your arbitrary filter function should obey $\mathcal{H}(-f) = \mathcal{H}(f)^*$. You can arrange this most easily by picking an $\mathcal{H}$ that is real and even in f.
- If your chosen $\mathcal{H}(f)$ has sharp vertical edges in it, then the *impulse response* of your filter (the output arising from a short impulse as input) will have damped "ringing" at frequencies corresponding to these edges. There is nothing wrong with this, but if you don't like it, then pick a smoother $\mathcal{H}(f)$. To get a first-hand look at the impulse response of your filter, just take the inverse FFT of your $\mathcal{H}(f)$. If you smooth all edges of the filter function over some number k of points, then the impulse response function of your filter will have a span on the order of a fraction $1/k$ of the whole data record.

```
do 19 j=1,m
    p(j)=p(j)/den                    Normalize the output.
enddo 19
return
END
```

CITED REFERENCES AND FURTHER READING:

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Harris, F.J. 1978, *Proceedings of the IEEE*, vol. 66, pp. 51–83. [2]

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), paper by P.D. Welch. [3]

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

13.5 Digital Filtering in the Time Domain

Suppose that you have a signal that you want to filter digitally. For example, perhaps you want to apply *high-pass* or *low-pass* filtering, to eliminate noise at low or high frequencies respectively; or perhaps the interesting part of your signal lies only in a certain frequency band, so that you need a *bandpass* filter. Or, if your measurements are contaminated by 60 Hz power-line interference, you may need a *notch filter* to remove only a narrow band around that frequency. This section speaks particularly about the case in which you have chosen to do such filtering in the time domain.

Before continuing, we hope you will reconsider this choice. Remember how convenient it is to filter in the Fourier domain. You just take your whole data record, FFT it, multiply the FFT output by a filter function $\mathcal{H}(f)$, and then do an inverse FFT to get back a filtered data set in time domain. Here is some additional background on the Fourier technique that you will want to take into account.

- Remember that you must define your filter function $\mathcal{H}(f)$ for both positive and negative frequencies, and that the magnitude of the frequency extremes is always the Nyquist frequency $1/(2\Delta)$, where Δ is the sampling interval. The magnitude of the smallest nonzero frequencies in the FFT is $\pm 1/(N\Delta)$, where N is the number of (complex) points in the FFT. The positive and negative frequencies to which this filter are applied are arranged in wrap-around order.
- If the measured data are real, and you want the filtered output also to be real, then your arbitrary filter function should obey $\mathcal{H}(-f) = \mathcal{H}(f)^*$. You can arrange this most easily by picking an $\mathcal{H}$ that is real and even in f.
- If your chosen $\mathcal{H}(f)$ has sharp vertical edges in it, then the *impulse response* of your filter (the output arising from a short impulse as input) will have damped "ringing" at frequencies corresponding to these edges. There is nothing wrong with this, but if you don't like it, then pick a smoother $\mathcal{H}(f)$. To get a first-hand look at the impulse response of your filter, just take the inverse FFT of your $\mathcal{H}(f)$. If you smooth all edges of the filter function over some number k of points, then the impulse response function of your filter will have a span on the order of a fraction $1/k$ of the whole data record.

- If your data set is too long to FFT all at once, then break it up into segments of any convenient size, as long as they are much longer than the impulse response function of the filter. Use zero-padding, if necessary.
- You should probably remove any trend from the data, by subtracting from it a straight line through the first and last points (i.e., make the first and last points equal to zero). If you are segmenting the data, then you can pick overlapping segments and use only the middle section of each, comfortably distant from edge effects.
- A digital filter is said to be *causal* or *physically realizable* if its output for a particular time-step depends only on inputs at that particular time-step or earlier. It is said to be *acausal* if its output can depend on both earlier and later inputs. Filtering in the Fourier domain is, in general, acausal, since the data are processed "in a batch," without regard to time ordering. Don't let this bother you! Acausal filters can generally give superior performance (e.g., less dispersion of phases, sharper edges, less asymmetric impulse response functions). People use causal filters not because they are better, but because some situations just don't allow access to out-of-time-order data. Time domain filters can, in principle, be either causal or acausal, but they are most often used in applications where physical realizability is a constraint. For this reason we will restrict ourselves to the causal case in what follows.

If you are still favoring time-domain filtering after all we have said, it is probably because you have a real-time application, for which you must process a continuous data stream and wish to output filtered values at the same rate as you receive raw data. Otherwise, it may be that the quantity of data to be processed is so large that you can afford only a very small number of floating operations on each data point and cannot afford even a modest-sized FFT (with a number of floating operations per data point several times the logarithm of the number of points in the data set or segment).

Linear Filters

The most general linear filter takes a sequence x_k of input points and produces a sequence y_n of output points by the formula

$$y_n = \sum_{k=0}^{M} c_k \, x_{n-k} + \sum_{j=1}^{N} d_j \, y_{n-j} \tag{13.5.1}$$

Here the $M+1$ coefficients c_k and the N coefficients d_j are fixed and define the filter response. The filter (13.5.1) produces each new output value from the current and M previous input values, and from its own N previous output values. If $N = 0$, so that there is no second sum in (13.5.1), then the filter is called *nonrecursive* or *finite impulse response (FIR)*. If $N \neq 0$, then it is called *recursive* or *infinite impulse response (IIR)*. (The term "IIR" connotes only that such filters are *capable* of having infinitely long impulse responses, not that their impulse response is necessarily long in a particular application. Typically the response of an IIR filter will drop off exponentially at late times, rapidly becoming negligible.)

The relation between the c_k's and d_j's and the filter response function $\mathcal{H}(f)$ is

$$\mathcal{H}(f) = \frac{\sum_{k=0}^{M} c_k e^{-2\pi i k (f\Delta)}}{1 - \sum_{j=1}^{N} d_j e^{-2\pi i j (f\Delta)}} \tag{13.5.2}$$

where Δ is, as usual, the sampling interval. The Nyquist interval corresponds to $f\Delta$ between $-1/2$ and $1/2$. For FIR filters the denominator of (13.5.2) is just unity.

Equation (13.5.2) tells how to determine $\mathcal{H}(f)$ from the c's and d's. To design a filter, though, we need a way of doing the inverse, getting a suitable set of c's and d's — as small a set as possible, to minimize the computational burden — from a desired $\mathcal{H}(f)$. Entire books are devoted to this issue. Like many other "inverse problems," it has no all-purpose

solution. One clearly has to make compromises, since $\mathcal{H}(f)$ is a full continuous function, while the short list of c's and d's represents only a few adjustable parameters. The subject of digital filter design concerns itself with the various ways of making these compromises. We cannot hope to give any sort of complete treatment of the subject. We can, however, sketch a couple of basic techniques to get you started. For further details, you will have to consult some specialized books (see references).

FIR (Nonrecursive) Filters

When the denominator in (13.5.2) is unity, the right-hand side is just a discrete Fourier transform. The transform is easily invertible, giving the desired small number of c_k coefficients in terms of the same small number of values of $\mathcal{H}(f_i)$ at some discrete frequencies f_i. This fact, however, is not very useful. The reason is that, for values of c_k computed in this way, $\mathcal{H}(f)$ will tend to oscillate wildly in between the discrete frequencies where it is pinned down to specific values.

A better strategy, and one which is the basis of several formal methods in the literature, is this: Start by pretending that you are willing to have a relatively large number of filter coefficients, that is, a relatively large value of M. Then $\mathcal{H}(f)$ can be fixed to desired values on a relatively fine mesh, and the M coefficients $c_k,\ k = 0, \ldots, M-1$ can be found by an FFT. Next, truncate (set to zero) most of the c_k's, leaving nonzero only the first, say, K, $(c_0, c_1, \ldots, c_{K-1})$ and last $K-1$, $(c_{M-K+1}, \ldots, c_{M-1})$. The last few c_k's are filter coefficients at *negative lag*, because of the wrap-around property of the FFT. But we don't want coefficients at negative lag. Therefore we cyclically shift the array of c_k's, to bring everything to positive lag. (This corresponds to introducing a time-delay into the filter.) Do this by copying the c_k's into a new array of length M in the following order:

$$(c_{M-K+1}, \ldots, c_{M-1},\ c_0,\ c_1, \ldots, c_{K-1},\ 0,\ 0, \ldots, 0) \tag{13.5.3}$$

To see if your truncation is acceptable, take the FFT of the array (13.5.3), giving an approximation to your original $\mathcal{H}(f)$. You will generally want to compare the *modulus* $|\mathcal{H}(f)|$ to your original function, since the time-delay will have introduced complex phases into the filter response.

If the new filter function is acceptable, then you are done and have a set of $2K-1$ filter coefficients. If it is not acceptable, then you can either (i) increase K and try again, or (ii) do something fancier to improve the acceptability for the same K. An example of something fancier is to modify the magnitudes (but not the phases) of the unacceptable $\mathcal{H}(f)$ to bring it more in line with your ideal, and then to FFT to get new c_k's. Once again set to zero all but the first $2K-1$ values of these (no need to cyclically shift since you have preserved the time-delaying phases), then inverse transform to get a new $\mathcal{H}(f)$, which will often be more acceptable. You can iterate this procedure. Note, however, that the procedure will not converge if your requirements for acceptability are more stringent than your $2K-1$ coefficients can handle.

The key idea, in other words, is to iterate between the space of coefficients and the space of functions $\mathcal{H}(f)$, until a Fourier conjugate pair that satisfies the imposed constraints *in both spaces* is found. A more formal technique for this kind of iteration is the *Remes Exchange Algorithm* which produces the best Chebyshev approximation to a given desired frequency response with a fixed number of filter coefficients (cf. §5.13).

IIR (Recursive) Filters

Recursive filters, whose output at a given time depends both on the current and previous inputs and on previous outputs, can generally have performance that is superior to nonrecursive filters with the same total number of coefficients (or same number of floating operations per input point). The reason is fairly clear by inspection of (13.5.2): A nonrecursive filter has a frequency response that is a polynomial in the variable $1/z$, where

$$z \equiv e^{2\pi i (f\Delta)} \tag{13.5.4}$$

By contrast, a recursive filter's frequency response is a *rational function* in $1/z$. The class of rational functions is especially good at fitting functions with sharp edges or narrow features, and most desired filter functions are in this category.

Nonrecursive filters are always stable. If you turn off the sequence of incoming x_i's, then after no more than M steps the sequence of y_j's produced by (13.5.1) will also turn off. Recursive filters, feeding as they do on their own output, are not necessarily stable. If the coefficients d_j are badly chosen, a recursive filter can have exponentially growing, so-called *homogeneous*, modes, which become huge even after the input sequence has been turned off. This is not good. The problem of designing recursive filters, therefore, is not just an inverse problem; it is an inverse problem with an additional stability constraint.

How do you tell if the filter (13.5.1) is stable for a given set of c_k and d_j coefficients? Stability depends only on the d_j's. The filter is stable if and only if all N complex roots of the *characteristic polynomial* equation

$$z^N - \sum_{j=1}^{N} d_j z^{N-j} = 0 \tag{13.5.5}$$

are inside the unit circle, i.e., satisfy

$$|z| \le 1 \tag{13.5.6}$$

The various methods for constructing stable recursive filters again form a subject area for which you will need more specialized books. One very useful technique, however, is the *bilinear transformation method*. For this topic we define a new variable w that reparametrizes the frequency f,

$$w \equiv \tan[\pi(f\Delta)] = i\left(\frac{1 - e^{2\pi i(f\Delta)}}{1 + e^{2\pi i(f\Delta)}}\right) = i\left(\frac{1-z}{1+z}\right) \tag{13.5.7}$$

Don't be fooled by the i's in (13.5.7). This equation maps real frequencies f into real values of w. In fact, it maps the Nyquist interval $-\frac{1}{2} < f\Delta < \frac{1}{2}$ onto the real w axis $-\infty < w < +\infty$. The inverse equation to (13.5.7) is

$$z = e^{2\pi i(f\Delta)} = \frac{1 + iw}{1 - iw} \tag{13.5.8}$$

In reparametrizing f, w also reparametrizes z, of course. Therefore, the condition for stability (13.5.5)–(13.5.6) can be rephrased in terms of w: If the filter response $\mathcal{H}(f)$ is written as a function of w, then the filter is stable if and only if the poles of the filter function (zeros of its denominator) are all in the upper half complex plane,

$$\mathrm{Im}(w) \ge 0 \tag{13.5.9}$$

The idea of the bilinear transformation method is that instead of specifying your desired $\mathcal{H}(f)$, you specify only its desired modulus square, $|\mathcal{H}(f)|^2 = \mathcal{H}(f)\mathcal{H}(f)^* = \mathcal{H}(f)\mathcal{H}(-f)$. Pick this to be approximated by some rational function in w^2. Then find all the poles of this function in the w complex plane. Every pole in the lower half-plane will have a corresponding pole in the upper half-plane, by symmetry. The idea is to form a product only of the factors with good poles, ones in the upper half-plane. This product is your *stably realizable* $\mathcal{H}(f)$. Now substitute equation (13.5.7) to write the function as a rational function in z, and compare with equation (13.5.2) to read off the c's and d's.

The procedure becomes clearer when we go through an example. Suppose we want to design a simple bandpass filter, whose lower cutoff frequency corresponds to a value $w = a$, and whose upper cutoff frequency corresponds to a value $w = b$, with a and b both positive numbers. A simple rational function that accomplishes this is

$$|\mathcal{H}(f)|^2 = \left(\frac{w^2}{w^2 + a^2}\right)\left(\frac{b^2}{w^2 + b^2}\right) \tag{13.5.10}$$

This function does not have a very sharp cutoff, but it is illustrative of the more general case. To obtain sharper edges, one could take the function (13.5.10) to some positive integer

power, or, equivalently, run the data sequentially through some number of copies of the filter that we will obtain from (13.5.10).

The poles of (13.5.10) are evidently at $w = \pm ia$ and $w = \pm ib$. Therefore the stably realizable $\mathcal{H}(f)$ is

$$\mathcal{H}(f) = \left(\frac{w}{w-ia}\right)\left(\frac{ib}{w-ib}\right) = \frac{\left(\frac{1-z}{1+z}\right)b}{\left[\left(\frac{1-z}{1+z}\right)-a\right]\left[\left(\frac{1-z}{1+z}\right)-b\right]} \tag{13.5.11}$$

We put the i in the numerator of the second factor in order to end up with real-valued coefficients. If we multiply out all the denominators, (13.5.11) can be rewritten in the form

$$\mathcal{H}(f) = \frac{-\frac{b}{(1+a)(1+b)} + \frac{b}{(1+a)(1+b)}z^{-2}}{1 - \frac{(1+a)(1-b)+(1-a)(1+b)}{(1+a)(1+b)}z^{-1} + \frac{(1-a)(1-b)}{(1+a)(1+b)}z^{-2}} \tag{13.5.12}$$

from which one reads off the filter coefficients for equation (13.5.1),

$$\begin{aligned} c_0 &= -\frac{b}{(1+a)(1+b)} \\ c_1 &= 0 \\ c_2 &= \frac{b}{(1+a)(1+b)} \\ d_1 &= \frac{(1+a)(1-b)+(1-a)(1+b)}{(1+a)(1+b)} \\ d_2 &= -\frac{(1-a)(1-b)}{(1+a)(1+b)} \end{aligned} \tag{13.5.13}$$

This completes the design of the bandpass filter.

Sometimes you can figure out how to construct directly a rational function in w for $\mathcal{H}(f)$, rather than having to start with its modulus square. The function that you construct has to have its poles only in the upper half-plane, for stability. It should also have the property of going into its own complex conjugate if you substitute $-w$ for w, so that the filter coefficients will be real.

For example, here is a function for a notch filter, designed to remove only a narrow frequency band around some fiducial frequency $w = w_0$, where w_0 is a positive number,

$$\begin{aligned} \mathcal{H}(f) &= \left(\frac{w-w_0}{w-w_0-i\epsilon w_0}\right)\left(\frac{w+w_0}{w+w_0-i\epsilon w_0}\right) \\ &= \frac{w^2-w_0^2}{(w-i\epsilon w_0)^2-w_0^2} \end{aligned} \tag{13.5.14}$$

In (13.5.14) the parameter ϵ is a small positive number that is the desired width of the notch, as a fraction of w_0. Going through the arithmetic of substituting z for w gives the filter coefficients

$$\begin{aligned} c_0 &= \frac{1+w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\ c_1 &= -2\frac{1-w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\ c_2 &= \frac{1+w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\ d_1 &= 2\frac{1-\epsilon^2 w_0^2-w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\ d_2 &= -\frac{(1-\epsilon w_0)^2+w_0^2}{(1+\epsilon w_0)^2+w_0^2} \end{aligned} \tag{13.5.15}$$

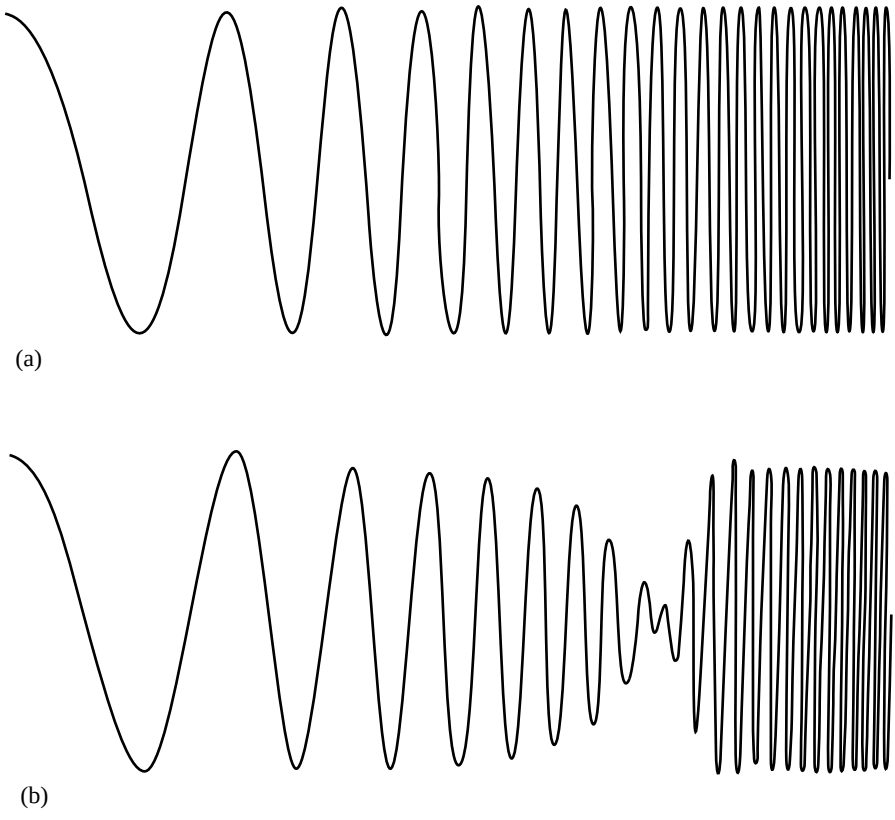


Figure 13.5.1. (a) A "chirp," or signal whose frequency increases continuously with time. (b) Same signal after it has passed through the notch filter (13.5.15). The parameter ϵ is here 0.2.

Figure 13.5.1 shows the results of using a filter of the form (13.5.15) on a "chirp" input signal, one that glides upwards in frequency, crossing the notch frequency along the way.

While the bilinear transformation may seem very general, its applications are limited by some features of the resulting filters. The method is good at getting the general shape of the desired filter, and good where "flatness" is a desired goal. However, the nonlinear mapping between w and f makes it difficult to design to a desired shape for a cutoff, and may move cutoff frequencies (defined by a certain number of dB) from their desired places. Consequently, practitioners of the art of digital filter design reserve the bilinear transformation for specific situations, and arm themselves with a variety of other tricks. We suggest that you do likewise, as your projects demand.

CITED REFERENCES AND FURTHER READING:

Hamming, R.W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall).

Antoniou, A. 1979, *Digital Filters: Analysis and Design* (New York: McGraw-Hill).

Parks, T.W., and Burrus, C.S. 1987, *Digital Filter Design* (New York: Wiley).

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

Rice, J.R. 1964, *The Approximation of Functions* (Reading, MA: Addison-Wesley); also 1969, *op. cit.*, Vol. 2.

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

13.6 Linear Prediction and Linear Predictive Coding

We begin with a very general formulation that will allow us to make connections to various special cases. Let $\{y'_\alpha\}$ be a set of measured values for some underlying set of true values of a quantity y, denoted $\{y_\alpha\}$, related to these true values by the addition of random noise,

$$y'_\alpha = y_\alpha + n_\alpha \tag{13.6.1}$$

(compare equation 13.3.2, with a somewhat different notation). Our use of a Greek subscript to index the members of the set is meant to indicate that the data points are not necessarily equally spaced along a line, or even ordered: they might be "random" points in three-dimensional space, for example. Now, suppose we want to construct the "best" estimate of the true value of some particular point $y_\star$ as a linear combination of the known, noisy, values. Writing

$$y_\star = \sum_\alpha d_{\star\alpha} y'_\alpha + x_\star \tag{13.6.2}$$

we want to find coefficients $d_{\star\alpha}$ that minimize, in some way, the *discrepancy* $x_\star$. The coefficients $d_{\star\alpha}$ have a "star" subscript to indicate that they depend on the choice of point $y_\star$. Later, we might want to let $y_\star$ be one of the existing y_α's. In that case, our problem becomes one of optimal filtering or estimation, closely related to the discussion in §13.3. On the other hand, we might want $y_\star$ to be a completely new point. In that case, our problem will be one of *linear prediction.*

A natural way to minimize the discrepancy $x_\star$ is in the statistical mean square sense. If angle brackets denote statistical averages, then we seek $d_{\star\alpha}$'s that minimize

$$\begin{aligned} \langle x_\star^2 \rangle &= \left\langle \left[\sum_\alpha d_{\star\alpha}(y_\alpha + n_\alpha) - y_\star \right]^2 \right\rangle \\ &= \sum_{\alpha\beta} (\langle y_\alpha y_\beta \rangle + \langle n_\alpha n_\beta \rangle) d_{\star\alpha} d_{\star\beta} - 2 \sum_\alpha \langle y_\star y_\alpha \rangle \, d_{\star\alpha} + \langle y_\star^2 \rangle \end{aligned} \tag{13.6.3}$$

Here we have used the fact that noise is uncorrelated with signal, e.g., $\langle n_\alpha y_\beta \rangle = 0$. The quantities $\langle y_\alpha y_\beta \rangle$ and $\langle y_\star y_\alpha \rangle$ describe the autocorrelation structure of the underlying data. We have already seen an analogous expression, (13.2.2), for the case of equally spaced data points on a line; we will meet correlation several times again in its statistical sense in Chapters 14 and 15. The quantities $\langle n_\alpha n_\beta \rangle$ describe the autocorrelation properties of the noise. Often, for point-to-point uncorrelated noise, we have $\langle n_\alpha n_\beta \rangle = \langle n_\alpha^2 \rangle \, \delta_{\alpha\beta}$. It is convenient to think of the various correlation quantities as comprising matrices and vectors,

$$\phi_{\alpha\beta} \equiv \langle y_\alpha y_\beta \rangle \qquad \phi_{\star\alpha} \equiv \langle y_\star y_\alpha \rangle \qquad \eta_{\alpha\beta} \equiv \langle n_\alpha n_\beta \rangle \quad \text{or} \quad \langle n_\alpha^2 \rangle \, \delta_{\alpha\beta} \tag{13.6.4}$$

Setting the derivative of equation (13.6.3) with respect to the $d_{\star\alpha}$'s equal to zero, one readily obtains the set of linear equations,

$$\sum_\beta [\phi_{\alpha\beta} + \eta_{\alpha\beta}] \, d_{\star\beta} = \phi_{\star\alpha} \tag{13.6.5}$$

13.6 Linear Prediction and Linear Predictive Coding

We begin with a very general formulation that will allow us to make connections to various special cases. Let $\{y'_\alpha\}$ be a set of measured values for some underlying set of true values of a quantity y, denoted $\{y_\alpha\}$, related to these true values by the addition of random noise,

$$y'_\alpha = y_\alpha + n_\alpha \tag{13.6.1}$$

(compare equation 13.3.2, with a somewhat different notation). Our use of a Greek subscript to index the members of the set is meant to indicate that the data points are not necessarily equally spaced along a line, or even ordered: they might be "random" points in three-dimensional space, for example. Now, suppose we want to construct the "best" estimate of the true value of some particular point $y_\star$ as a linear combination of the known, noisy, values. Writing

$$y_\star = \sum_\alpha d_{\star\alpha} y'_\alpha + x_\star \tag{13.6.2}$$

we want to find coefficients $d_{\star\alpha}$ that minimize, in some way, the *discrepancy* $x_\star$. The coefficients $d_{\star\alpha}$ have a "star" subscript to indicate that they depend on the choice of point $y_\star$. Later, we might want to let $y_\star$ be one of the existing y_α's. In that case, our problem becomes one of optimal filtering or estimation, closely related to the discussion in §13.3. On the other hand, we might want $y_\star$ to be a completely new point. In that case, our problem will be one of *linear prediction.*

A natural way to minimize the discrepancy $x_\star$ is in the statistical mean square sense. If angle brackets denote statistical averages, then we seek $d_{\star\alpha}$'s that minimize

$$\begin{aligned} \langle x_\star^2 \rangle &= \left\langle \left[\sum_\alpha d_{\star\alpha}(y_\alpha + n_\alpha) - y_\star \right]^2 \right\rangle \\ &= \sum_{\alpha\beta} (\langle y_\alpha y_\beta \rangle + \langle n_\alpha n_\beta \rangle) d_{\star\alpha} d_{\star\beta} - 2 \sum_\alpha \langle y_\star y_\alpha \rangle \, d_{\star\alpha} + \langle y_\star^2 \rangle \end{aligned} \tag{13.6.3}$$

Here we have used the fact that noise is uncorrelated with signal, e.g., $\langle n_\alpha y_\beta \rangle = 0$. The quantities $\langle y_\alpha y_\beta \rangle$ and $\langle y_\star y_\alpha \rangle$ describe the autocorrelation structure of the underlying data. We have already seen an analogous expression, (13.2.2), for the case of equally spaced data points on a line; we will meet correlation several times again in its statistical sense in Chapters 14 and 15. The quantities $\langle n_\alpha n_\beta \rangle$ describe the autocorrelation properties of the noise. Often, for point-to-point uncorrelated noise, we have $\langle n_\alpha n_\beta \rangle = \langle n_\alpha^2 \rangle \, \delta_{\alpha\beta}$. It is convenient to think of the various correlation quantities as comprising matrices and vectors,

$$\phi_{\alpha\beta} \equiv \langle y_\alpha y_\beta \rangle \qquad \phi_{\star\alpha} \equiv \langle y_\star y_\alpha \rangle \qquad \eta_{\alpha\beta} \equiv \langle n_\alpha n_\beta \rangle \quad \text{or} \quad \langle n_\alpha^2 \rangle \, \delta_{\alpha\beta} \tag{13.6.4}$$

Setting the derivative of equation (13.6.3) with respect to the $d_{\star\alpha}$'s equal to zero, one readily obtains the set of linear equations,

$$\sum_\beta [\phi_{\alpha\beta} + \eta_{\alpha\beta}] \, d_{\star\beta} = \phi_{\star\alpha} \tag{13.6.5}$$

If we write the solution as a matrix inverse, then the estimation equation (13.6.2) becomes, omitting the minimized discrepancy $x_\star$,

$$y_\star \approx \sum_{\alpha\beta} \phi_{\star\alpha} \left[\phi_{\mu\nu} + \eta_{\mu\nu}\right]^{-1}_{\alpha\beta} y'_\beta \tag{13.6.6}$$

From equations (13.6.3) and (13.6.5) one can also calculate the expected mean square value of the discrepancy at its minimum, denoted $\left\langle x_\star^2 \right\rangle_0$,

$$\left\langle x_\star^2 \right\rangle_0 = \left\langle y_\star^2 \right\rangle - \sum_\beta d_{\star\beta}\phi_{\star\beta} = \left\langle y_\star^2 \right\rangle - \sum_{\alpha\beta} \phi_{\star\alpha} \left[\phi_{\mu\nu} + \eta_{\mu\nu}\right]^{-1}_{\alpha\beta} \phi_{\star\beta} \tag{13.6.7}$$

A final general result tells how much the mean square discrepancy $\left\langle x_\star^2 \right\rangle$ is increased if we use the estimation equation (13.6.2) not with the best values $d_{\star\beta}$, but with some other values $\widehat{d}_{\star\beta}$. The above equations then imply

$$\left\langle x_\star^2 \right\rangle = \left\langle x_\star^2 \right\rangle_0 + \sum_{\alpha\beta} (\widehat{d}_{\star\alpha} - d_{\star\alpha}) \left[\phi_{\alpha\beta} + \eta_{\alpha\beta}\right] (\widehat{d}_{\star\beta} - d_{\star\beta}) \tag{13.6.8}$$

Since the second term is a pure quadratic form, we see that the increase in the discrepancy is only second order in any error made in estimating the $d_{\star\beta}$'s.

Connection to Optimal Filtering

If we change "star" to a Greek index, say γ, then the above formulas describe optimal filtering, generalizing the discussion of §13.3. One sees, for example, that if the noise amplitudes n_α go to zero, so likewise do the noise autocorrelations $\eta_{\alpha\beta}$, and, canceling a matrix times its inverse, equation (13.6.6) simply becomes $y_\gamma = y'_\gamma$. Another special case occurs if the matrices $\phi_{\alpha\beta}$ and $\eta_{\alpha\beta}$ are diagonal. In that case, equation (13.6.6) becomes

$$y_\gamma = \frac{\phi_{\gamma\gamma}}{\phi_{\gamma\gamma} + \eta_{\gamma\gamma}} y'_\gamma \tag{13.6.9}$$

which is readily recognizable as equation (13.3.6) with $S^2 \to \phi_{\gamma\gamma}$, $N^2 \to \eta_{\gamma\gamma}$. What is going on is this: For the case of equally spaced data points, and in the Fourier domain, autocorrelations become simply squares of Fourier amplitudes (Wiener-Khinchin theorem, equation 12.0.12), and the optimal filter can be constructed algebraically, as equation (13.6.9), without inverting any matrix.

More generally, in the time domain, or any other domain, an optimal filter (one that minimizes the square of the discrepancy from the underlying true value in the presence of measurement noise) can be constructed by estimating the autocorrelation matrices $\phi_{\alpha\beta}$ and $\eta_{\alpha\beta}$, and applying equation (13.6.6) with $\star \to \gamma$. (Equation 13.6.8 is in fact the basis for the §13.3's statement that even crude optimal filtering can be quite effective.)

Linear Prediction

Classical *linear prediction* specializes to the case where the data points y_β are equally spaced along a line, y_i, $i = 1, 2, \ldots, N$, and we want to use M consecutive values of y_i to predict an $M + 1$st. Stationarity is assumed. That is, the autocorrelation $\langle y_j y_k \rangle$ is assumed to depend only on the difference $|j - k|$, and not on j or k individually, so that the autocorrelation ϕ has only a single index,

$$\phi_j \equiv \langle y_i y_{i+j} \rangle \approx \frac{1}{N-j} \sum_{i=1}^{N-j} y_i y_{i+j} \tag{13.6.10}$$

Here, the approximate equality shows one way to use the actual data set values to estimate the autocorrelation components. (In fact, there is a better way to make these estimates; see below.) In the situation described, the estimation equation (13.6.2) is

$$y_n = \sum_{j=1}^{M} d_j y_{n-j} + x_n \tag{13.6.11}$$

(compare equation 13.5.1) and equation (13.6.5) becomes the set of M equations for the M unknown d_j's, now called the *linear prediction (LP) coefficients*,

$$\sum_{j=1}^{M} \phi_{|j-k|}\, d_j = \phi_k \qquad (k = 1, \ldots, M) \tag{13.6.12}$$

Notice that while noise is not explicitly included in the equations, it is properly accounted for, *if* it is point-to-point uncorrelated: ϕ_0, as estimated by equation (13.6.10) using *measured* values y_i', actually estimates the diagonal part of $\phi_{\alpha\alpha} + \eta_{\alpha\alpha}$, above. The mean square discrepancy $\langle x_n^2 \rangle$ is estimated by equation (13.6.7) as

$$\langle x_n^2 \rangle = \phi_0 - \phi_1 d_1 - \phi_2 d_2 - \cdots - \phi_M d_M \tag{13.6.13}$$

To use linear prediction, we first compute the d_j's, using equations (13.6.10) and (13.6.12). We then calculate equation (13.6.13) or, more concretely, apply (13.6.11) to the known record to get an idea of how large are the discrepancies x_i. If the discrepancies are small, then we can continue applying (13.6.11) right on into the future, imagining the unknown "future" discrepancies x_i to be zero. In this application, (13.6.11) is a kind of extrapolation formula. In many situations, this extrapolation turns out to be vastly more powerful than any kind of simple polynomial extrapolation. (By the way, you should not confuse the terms "linear prediction" and "linear extrapolation"; the general functional form used by linear prediction is *much* more complex than a straight line, or even a low-order polynomial!)

However, to achieve its full usefulness, linear prediction must be constrained in one additional respect: One must take additional measures to guarantee its *stability*. Equation (13.6.11) is a special case of the general linear filter (13.5.1). The condition that (13.6.11) be stable as a linear predictor is precisely that given in equations (13.5.5) and (13.5.6), namely that the characteristic polynomial

$$z^N - \sum_{j=1}^{N} d_j z^{N-j} = 0 \tag{13.6.14}$$

have all N of its roots inside the unit circle,

$$|z| \leq 1 \tag{13.6.15}$$

There is no guarantee that the coefficients produced by equation (13.6.12) will have this property. If the data contain many oscillations without any particular trend towards increasing or decreasing amplitude, then the complex roots of (13.6.14) will generally all be rather close to the unit circle. The finite length of the data set will cause some of these roots to be inside the unit circle, others outside. In some applications, where the resulting instabilities are slowly growing and the linear prediction is not pushed too far, it is best to use the "unmassaged" LP coefficients that come directly out of (13.6.12). For example, one might be extrapolating to fill a short gap in a data set; then one might extrapolate both forwards across the gap and backwards from the data beyond the gap. If the two extrapolations agree tolerably well, then instability is not a problem.

When instability *is* a problem, you have to "massage" the LP coefficients. You do this by (i) solving (numerically) equation (13.6.14) for its N complex roots; (ii) moving the roots to where you think they ought to be inside or on the unit circle; (iii) reconstituting the now-modified LP coefficients. You may think that step (ii) sounds a little vague. It is. There is no "best" procedure. If you think that your signal is truly a sum of undamped sine and cosine waves (perhaps with incommensurate periods), then you will want simply to move each root z_i onto the unit circle,

$$z_i \;\rightarrow\; z_i / |z_i| \tag{13.6.16}$$

In other circumstances it may seem appropriate to reflect a bad root across the unit circle

$$z_i \;\rightarrow\; 1/z_i{}^* \tag{13.6.17}$$

This alternative has the property that it preserves the amplitude of the output of (13.6.11) when it is driven by a sinusoidal set of x_i's. It assumes that (13.6.12) has correctly identified the spectral width of a resonance, but only slipped up on identifying its time sense so that signals that should be damped as time proceeds end up growing in amplitude. The choice between (13.6.16) and (13.6.17) sometimes might as well be based on voodoo. We prefer (13.6.17).

Also magical is the choice of M, the number of LP coefficients to use. You should choose M to be as small as works for you, that is, you should choose it by experimenting with your data. Try $M = 5, 10, 20, 40$. If you need larger M's than this, be aware that the procedure of "massaging" all those complex roots is quite sensitive to roundoff error. Use double precision.

Linear prediction is especially successful at extrapolating signals that are smooth and oscillatory, though not necessarily periodic. In such cases, linear prediction often extrapolates accurately through *many cycles* of the signal. By contrast, polynomial extrapolation in general becomes seriously inaccurate after at most a cycle or two. A prototypical example of a signal that can successfully be linearly predicted is the height of ocean tides, for which the fundamental 12-hour period is modulated in phase and amplitude over the course of the month and year, and for which local

hydrodynamic effects may make even one cycle of the curve look rather different in shape from a sine wave.

We already remarked that equation (13.6.10) is not necessarily the best way to estimate the covariances ϕ_k from the data set. In fact, results obtained from linear prediction are remarkably sensitive to exactly how the ϕ_k's are estimated. One particularly good method is due to Burg [1], and involves a recursive procedure for increasing the order M by one unit at a time, at each stage re-estimating the coefficients d_j, $j = 1, \ldots, M$ so as to minimize the residual in equation (13.6.13). Although further discussion of the Burg method is beyond our scope here, the method is implemented in the following routine [1,2] for estimating the LP coefficients d_j of a data set.

```
SUBROUTINE memcof(data,n,m,xms,d)
INTEGER m,n,MMAX,NMAX
REAL xms,d(m),data(n)
PARAMETER (MMAX=60,NMAX=2000)
    Given a real vector of data(1:n), and given m, this routine returns m linear prediction
    coefficients as d(1:m), and returns the mean square discrepancy as xms.
INTEGER i,j,k
REAL denom,p,pneum,wk1(NMAX),wk2(NMAX),wkm(MMAX)
if (m.gt.MMAX.or.n.gt.NMAX) pause 'workspace too small in memcof'
p=0.
do 11 j=1,n
    p=p+data(j)**2
enddo 11
xms=p/n
wk1(1)=data(1)
wk2(n-1)=data(n)
do 12 j=2,n-1
    wk1(j)=data(j)
    wk2(j-1)=data(j)
enddo 12
do 17 k=1,m
    pneum=0.
    denom=0.
    do 13 j=1,n-k
        pneum=pneum+wk1(j)*wk2(j)
        denom=denom+wk1(j)**2+wk2(j)**2
    enddo 13
    d(k)=2.*pneum/denom
    xms=xms*(1.-d(k)**2)
    do 14 i=1,k-1
        d(i)=wkm(i)-d(k)*wkm(k-i)
    enddo 14
      The algorithm is recursive, building up the answer for larger and larger values of m until
      the desired value is reached. At this point in the algorithm, one could return the vector
      d and scalar xms for a set of LP coefficients with k (rather than m) terms.
    if(k.eq.m)return
    do 15 i=1,k
        wkm(i)=d(i)
    enddo 15
    do 16 j=1,n-k-1
        wk1(j)=wk1(j)-wkm(k)*wk2(j)
        wk2(j)=wk2(j+1)-wkm(k)*wk1(j+1)
    enddo 16
enddo 17
pause 'never get here in memcof'
END
```

Here are procedures for rendering the LP coefficients stable (if you choose to do so), and for extrapolating a data set by linear prediction, using the original or massaged LP coefficients. The routine zroots (§9.5) is used to find all complex roots of a polynomial.

```
      SUBROUTINE fixrts(d,m)
      INTEGER m,MMAX
      REAL d(m)
      PARAMETER (MMAX=100)          Largest expected value of m.
C     USES zroots
        Given the LP coefficients d(1:m), this routine finds all roots of the characteristic polynomial
        (13.6.14), reflects any roots that are outside the unit circle back inside, and then returns
        a modified set of coefficients d(1:m).
      INTEGER i,j
      LOGICAL polish
      COMPLEX a(MMAX),roots(MMAX)
      a(m+1)=cmplx(1.,0.)
      do 11 j=m,1,-1                Set up complex coefficients for polynomial root finder.
        a(j)=cmplx(-d(m+1-j),0.)
      enddo 11
      polish=.true.
      call zroots(a,m,roots,polish)          Find all the roots.
      do 12 j=1,m                            Look for a...
        if(abs(roots(j)).gt.1.)then          root outside the unit circle,
          roots(j)=1./conjg(roots(j))        and reflect it back inside.
        endif
      enddo 12
      a(1)=-roots(1)                Now reconstruct the polynomial coefficients,
      a(2)=cmplx(1.,0.)
      do 14 j=2,m                   by looping over the roots
        a(j+1)=cmplx(1.,0.)
        do 13 i=j,2,-1              and synthetically multiplying.
          a(i)=a(i-1)-roots(j)*a(i)
        enddo 13
        a(1)=-roots(j)*a(1)
      enddo 14
      do 15 j=1,m                   The polynomial coefficients are guaranteed to be real,
        d(m+1-j)=-real(a(j))        so we need only return the real part as new LP coefficients.
      enddo 15
      return
      END
```

```
      SUBROUTINE predic(data,ndata,d,m,future,nfut)
      INTEGER ndata,nfut,m,MMAX
      REAL d(m),data(ndata),future(nfut)
      PARAMETER (MMAX=100)
        Given data(1:ndata), and given the data's LP coefficients d(1:m), this routine applies
        equation (13.6.11) to predict the next nfut data points, which it returns in the array
        future(1:nfut). Note that the routine references only the last m values of data, as
        initial values for the prediction.
        Parameter: MMAX is the largest expected value of m.
      INTEGER j,k
      REAL discrp,sum,reg(MMAX)
      do 11 j=1,m
        reg(j)=data(ndata+1-j)
      enddo 11
      do 14 j=1,nfut
        discrp=0.                   This is where you would put in a known discrepancy if you
        sum=discrp                    were reconstructing a function by linear predictive coding
        do 12 k=1,m                   rather than extrapolating a function by linear prediction.
                                      See text.
```

```
        sum=sum+d(k)*reg(k)
    enddo 12
    do 13 k=m,2,-1                [If you want to implement circular arrays, you can avoid this
        reg(k)=reg(k-1)               shifting of coefficients!]
    enddo 13
    reg(1)=sum
    future(j)=sum
enddo 14
return
END
```

Removing the Bias in Linear Prediction

You might expect that the sum of the d_j's in equation (13.6.11) (or, more generally, in equation 13.6.2) should be 1, so that (e.g.) adding a constant to all the data points y_i yields a prediction that is increased by the same constant. However, the d_j's do not sum to 1 but, in general, to a value slightly less than one. This fact reveals a subtle point, that the estimator of classical linear prediction is not *unbiased*, even though it does minimize the mean square discrepancy. At any place where the measured autocorrelation does not imply a better estimate, the equations of linear prediction tend to predict a value that tends towards zero.

Sometimes, that is just what you want. If the process that generates the y_i's in fact has zero mean, then zero is the best guess absent other information. At other times, however, this behavior is unwarranted. If you have data that show only small variations around a positive value, you don't want linear predictions that droop towards zero.

Often it is a workable approximation to subtract the mean off your data set, perform the linear prediction, and then add the mean back. This procedure contains the germ of the correct solution; but the simple arithmetic mean is not quite the correct constant to subtract. In fact, an unbiased estimator is obtained by subtracting from every data point an autocorrelation-weighted mean defined by [3,4]

$$\overline{y} \equiv \sum_{\beta} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta}\, y_\beta \Bigg/ \sum_{\alpha\beta} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta} \tag{13.6.18}$$

With this subtraction, the sum of the LP coefficients should be unity, up to roundoff and differences in how the ϕ_k's are estimated.

Linear Predictive Coding (LPC)

A different, though related, method to which the formalism above can be applied is the "compression" of a sampled signal so that it can be stored more compactly. The original form should be *exactly* recoverable from the compressed version. Obviously, compression can be accomplished only if there is redundancy in the signal. Equation (13.6.11) describes one kind of redundancy: It says that the signal, except for a small discrepancy, is predictable from its previous values and from a small number of LP coefficients. Compression of a signal by the use of (13.6.11) is thus called *linear predictive coding*, or *LPC*.

The basic idea of LPC (in its simplest form) is to record as a compressed file (i) the number of LP coefficients M, (ii) their M values, e.g., as obtained by `memcof`,

(iii) the first M data points, and then (iv) for each subsequent data point only its residual discrepancy x_i (equation 13.6.1). When you are creating the compressed file, you find the residual by applying (13.6.1) to the previous M points, subtracting the sum from the actual value of the current point. When you are reconstructing the original file, you add the residual back in, at the point indicated in the routine `predic`.

It may not be obvious why there is any compression at all in this scheme. After all, we are storing one value of residual per data point! Why not just store the original data point? The answer depends on the relative sizes of the numbers involved. The residual is obtained by subtracting two very nearly equal numbers (the data and the linear prediction). Therefore, the discrepancy typically has only a very small number of nonzero bits. These can be stored in a compressed file. How do you do it in a high-level language? Here is one way: Scale your data to have integer values, say between $+1000000$ and -1000000 (supposing that you need six significant figures). Modify equation (13.6.1) by enclosing the sum term in an "integer part of" operator. The discrepancy will now, by definition, be an integer. Experiment with different values of M, to find LP coefficients that make the range of the discrepancy as small as you can. If you can get to within a range of ± 127 (and in our experience this is not at all difficult) then you can write it to a file as a single byte. This is a compression factor of 4, compared to 4-byte integer or floating formats.

Notice that the LP coefficients are computed using the *quantized* data, and that the discrepancy is also quantized, i.e., quantization is done both outside and inside the LPC loop. If you are careful in following this prescription, then, apart from the initial quantization of the data, you will not introduce even a single bit of roundoff error into the compression-reconstruction process: While the evaluation of the sum in (13.6.11) may have roundoff errors, the residual that you store is the value which, when added back to the sum, gives *exactly* the original (quantized) data value. Notice also that you do not need to massage the LP coefficients for stability; by adding the residual back in to each point, you never depart from the original data, so instabilities cannot grow. There is therefore no need for `fixrts`, above.

Look at §20.4 to learn about *Huffman coding*, which will further compress the residuals by taking advantage of the fact that smaller values of discrepancy will occur more often than larger values. A very primitive version of Huffman coding would be this: If most of the discrepancies are in the range ± 127, but an occasional one is outside, then reserve the value 127 to mean "out of range," and then record on the file (immediately following the 127) a full-word value of the out-of-range discrepancy. §20.4 explains how to do much better.

There are many variant procedures that all fall under the rubric of LPC.

- If the spectral character of the data is time-variable, then it is best not to use a single set of LP coefficients for the whole data set, but rather to partition the data into segments, computing and storing different LP coefficients for each segment.
- If the data are really well characterized by their LP coefficients, and you can tolerate some small amount of error, then don't bother storing all of the residuals. Just do linear prediction until you are outside of tolerances, then reinitialize (using M sequential stored residuals) and continue predicting.
- In some applications, most notably speech synthesis, one cares only about the spectral content of the reconstructed signal, not the relative phases. In this case, one need not store any starting values at all, but only the

LP coefficients for each segment of the data. The output is reconstructed by driving these coefficients with initial conditions consisting of all zeros except for one nonzero spike. A speech synthesizer chip may have of order 10 LP coefficients, which change perhaps 20 to 50 times per second.

- Some people believe that it is interesting to analyze a signal by LPC, even when the residuals x_i are *not* small. The x_i's are then interpreted as the underlying "input signal" which, when filtered through the all-poles filter defined by the LP coefficients (see §13.7), produces the observed "output signal." LPC reveals simultaneously, it is said, the nature of the filter *and* the particular input that is driving it. We are skeptical of these applications; the literature, however, is full of extravagant claims.

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), especially the paper by J. Makhoul (reprinted from *Proceedings of the IEEE*, vol. 63, p. 561, 1975).

Burg, J.P. 1968, reprinted in Childers, 1978. [1]

Anderson, N. 1974, reprinted in Childers, 1978. [2]

Cressie, N. 1991, in *Spatial Statistics and Digital Image Analysis* (Washington: National Academy Press). [3]

Press, W.H., and Rybicki, G.B. 1992, *Astrophysical Journal*, vol. 398, pp. 169–176. [4]

13.7 Power Spectrum Estimation by the Maximum Entropy (All Poles) Method

The FFT is not the only way to estimate the power spectrum of a process, nor is it necessarily the best way for all purposes. To see how one might devise another method, let us enlarge our view for a moment, so that it includes not only real frequencies in the Nyquist interval $-f_c < f < f_c$, but also the entire complex frequency plane. From that vantage point, let us transform the complex f-plane to a new plane, called the *z-transform plane* or *z-plane*, by the relation

$$z \equiv e^{2\pi i f \Delta} \tag{13.7.1}$$

where Δ is, as usual, the sampling interval in the time domain. Notice that the Nyquist interval on the real axis of the f-plane maps one-to-one onto the unit circle in the complex z-plane.

If we now compare (13.7.1) to equations (13.4.4) and (13.4.6), we see that the FFT power spectrum estimate (13.4.5) for any real sampled function $c_k \equiv c(t_k)$ can be written, except for normalization convention, as

$$P(f) = \left| \sum_{k=-N/2}^{N/2-1} c_k z^k \right|^2 \tag{13.7.2}$$

Of course, (13.7.2) is not the *true* power spectrum of the underlying function $c(t)$, but only an estimate. We can see in two related ways why the estimate is not likely to be exact. First, in the time domain, the estimate is based on only a finite range of the function $c(t)$ which may, for all we know, have continued from $t = -\infty$ to ∞. Second, in the z-plane of equation (13.7.2), the finite Laurent series offers, in general, only an approximation to a general analytic function of z. In fact, a formal expression for representing "true" power spectra (up to normalization) is

$$P(f) = \left| \sum_{k=-\infty}^{\infty} c_k z^k \right|^2 \tag{13.7.3}$$

LP coefficients for each segment of the data. The output is reconstructed by driving these coefficients with initial conditions consisting of all zeros except for one nonzero spike. A speech synthesizer chip may have of order 10 LP coefficients, which change perhaps 20 to 50 times per second.

- Some people believe that it is interesting to analyze a signal by LPC, even when the residuals x_i are *not* small. The x_i's are then interpreted as the underlying "input signal" which, when filtered through the all-poles filter defined by the LP coefficients (see §13.7), produces the observed "output signal." LPC reveals simultaneously, it is said, the nature of the filter *and* the particular input that is driving it. We are skeptical of these applications; the literature, however, is full of extravagant claims.

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), especially the paper by J. Makhoul (reprinted from *Proceedings of the IEEE*, vol. 63, p. 561, 1975).

Burg, J.P. 1968, reprinted in Childers, 1978. [1]

Anderson, N. 1974, reprinted in Childers, 1978. [2]

Cressie, N. 1991, in *Spatial Statistics and Digital Image Analysis* (Washington: National Academy Press). [3]

Press, W.H., and Rybicki, G.B. 1992, *Astrophysical Journal*, vol. 398, pp. 169–176. [4]

13.7 Power Spectrum Estimation by the Maximum Entropy (All Poles) Method

The FFT is not the only way to estimate the power spectrum of a process, nor is it necessarily the best way for all purposes. To see how one might devise another method, let us enlarge our view for a moment, so that it includes not only real frequencies in the Nyquist interval $-f_c < f < f_c$, but also the entire complex frequency plane. From that vantage point, let us transform the complex f-plane to a new plane, called the *z-transform plane* or *z-plane*, by the relation

$$z \equiv e^{2\pi i f \Delta} \tag{13.7.1}$$

where Δ is, as usual, the sampling interval in the time domain. Notice that the Nyquist interval on the real axis of the f-plane maps one-to-one onto the unit circle in the complex z-plane.

If we now compare (13.7.1) to equations (13.4.4) and (13.4.6), we see that the FFT power spectrum estimate (13.4.5) for any real sampled function $c_k \equiv c(t_k)$ can be written, except for normalization convention, as

$$P(f) = \left| \sum_{k=-N/2}^{N/2-1} c_k z^k \right|^2 \tag{13.7.2}$$

Of course, (13.7.2) is not the *true* power spectrum of the underlying function $c(t)$, but only an estimate. We can see in two related ways why the estimate is not likely to be exact. First, in the time domain, the estimate is based on only a finite range of the function $c(t)$ which may, for all we know, have continued from $t = -\infty$ to ∞. Second, in the z-plane of equation (13.7.2), the finite Laurent series offers, in general, only an approximation to a general analytic function of z. In fact, a formal expression for representing "true" power spectra (up to normalization) is

$$P(f) = \left| \sum_{k=-\infty}^{\infty} c_k z^k \right|^2 \tag{13.7.3}$$

This is an infinite Laurent series which depends on an infinite number of values c_k. Equation (13.7.2) is just one kind of analytic approximation to the analytic function of z represented by (13.7.3); the kind, in fact, that is implicit in the use of FFTs to estimate power spectra by periodogram methods. It goes under several names, including *direct method, all-zero model,* and *moving average (MA) model.* The term "all-zero" in particular refers to the fact that the model spectrum can have zeros in the z-plane, but not poles.

If we look at the problem of approximating (13.7.3) more generally it seems clear that we could do a better job with a rational function, one with a series of type (13.7.2) in both the numerator and the denominator. Less obviously, it turns out that there are some advantages in an approximation whose free parameters all lie in the *denominator*, namely,

$$P(f) \approx \frac{1}{\left| \sum_{k=-M/2}^{M/2} b_k z^k \right|^2} = \frac{a_0}{\left| 1 + \sum_{k=1}^{M} a_k z^k \right|^2} \tag{13.7.4}$$

Here the second equality brings in a new set of coefficients a_k's, which can be determined from the b_k's using the fact that z lies on the unit circle. The b_k's can be thought of as being determined by the condition that power series expansion of (13.7.4) agree with the first $M + 1$ terms of (13.7.3). In practice, as we shall see, one determines the b_k's or a_k's by another method.

The differences between the approximations (13.7.2) and (13.7.4) are not just cosmetic. They are approximations with very different character. Most notable is the fact that (13.7.4) can have *poles*, corresponding to infinite power spectral density, on the unit z-circle, i.e., at real frequencies in the Nyquist interval. Such poles can provide an accurate representation for underlying power spectra that have sharp, discrete "lines" or delta-functions. By contrast, (13.7.2) can have only zeros, not poles, at real frequencies in the Nyquist interval, and must thus attempt to fit sharp spectral features with, essentially, a polynomial. The approximation (13.7.4) goes under several names: *all-poles model, maximum entropy method (MEM), autoregressive model (AR).* We need only find out how to compute the coefficients a_0 and the a_k's from a data set, so that we can actually use (13.7.4) to obtain spectral estimates.

A pleasant surprise is that we already know how! Look at equation (13.6.11) for linear prediction. Compare it with linear filter equations (13.5.1) and (13.5.2), and you will see that, viewed as a filter that takes input x's into output y's, linear prediction has a filter function

$$\mathcal{H}(f) = \frac{1}{1 - \sum_{j=1}^{N} d_j z^j} \tag{13.7.5}$$

Thus, the power spectrum of the y's should be equal to the power spectrum of the x's multiplied by $|\mathcal{H}(f)|^2$. Now let us think about what the spectrum of the input x's is, when they are residual discrepancies from linear prediction. Although we will not prove it formally, it is intuitively believable that the x's are independently random and therefore have a flat (white noise) spectrum. (Roughly speaking, any residual correlations left in the x's would have allowed a more accurate linear prediction, and would have been removed.) The overall normalization of this flat spectrum is just the mean square amplitude of the x's. But this is exactly the quantity computed in equation (13.6.13) and returned by the routine `memcof` as `xms`. Thus, the coefficients a_0 and a_k in equation (13.7.4) are related to the LP coefficients returned by `memcof` simply by

$$a_0 = \texttt{xms} \qquad a_k = -\texttt{d(}k\texttt{)}, \quad k = 1, \ldots, M \tag{13.7.6}$$

There is also another way to describe the relation between the a_k's and the autocorrelation components ϕ_k. The Wiener-Khinchin theorem (12.0.12) says that the Fourier transform of the autocorrelation is equal to the power spectrum. In z-transform language, this Fourier transform is just a Laurent series in z. The equation that is to be satisfied by the coefficients in equation (13.7.4) is thus

$$\frac{a_0}{\left| 1 + \sum_{k=1}^{M} a_k z^k \right|^2} \approx \sum_{j=-M}^{M} \phi_j z^j \tag{13.7.7}$$

The approximately equal sign in (13.7.7) has a somewhat special interpretation. It means that the series expansion of the left-hand side is supposed to agree with the right-hand side term by term from z^{-M} to z^{M}. Outside this range of terms, the right-hand side is obviously zero, while the left-hand side will still have nonzero terms. Notice that M, the number of coefficients in the approximation on the left-hand side, can be any integer up to N, the total number of autocorrelations available. (In practice, one often chooses M much smaller than N.) M is called the *order* or *number of poles* of the approximation.

Whatever the chosen value of M, the series expansion of the left-hand side of (13.7.7) defines a certain sort of *extrapolation* of the autocorrelation function to lags larger than M, in fact even to lags larger than N, i.e., *larger than the run of data can actually measure*. It turns out that this particular extrapolation can be shown to have, among all possible extrapolations, the maximum *entropy* in a definable information-theoretic sense. Hence the name *maximum entropy method*, or MEM. The maximum entropy property has caused MEM to acquire a certain "cult" popularity; one sometimes hears that it gives an intrinsically "better" estimate than is given by other methods. Don't believe it. MEM has the very cute property of being able to fit sharp spectral features, but there is nothing else magical about its power spectrum estimates.

The operations count in `memcof` scales as the product of N (the number of data points) and M (the desired order of the MEM approximation). If M were chosen to be as large as N, then the method would be much slower than the $N \log N$ FFT methods of the previous section. In practice, however, one usually wants to limit the order (or number of poles) of the MEM approximation to a few times the number of sharp spectral features that one desires it to fit. With this restricted number of poles, the method will smooth the spectrum somewhat, but this is often a desirable property. While exact values depend on the application, one might take $M = 10$ or 20 or 50 for $N = 1000$ or 10000. In that case MEM estimation is not much slower than FFT estimation.

We feel obliged to warn you that `memcof` can be a bit quirky at times. If the number of poles or number of data points is too large, roundoff error can be a problem, even in double precision. With "peaky" data (i.e., data with extremely sharp spectral features), the algorithm may suggest split peaks even at modest orders, and the peaks may shift with the phase of the sine wave. Also, with noisy input functions, if you choose too high an order, you will find spurious peaks galore! Some experts recommend the use of this algorithm in conjunction with more conservative methods, like periodograms, to help choose the correct model order, and to avoid getting too fooled by spurious spectral features. MEM can be finicky, but it can also do remarkable things. We recommend that you try it out, cautiously, on your own problems. We now turn to the evaluation of the MEM spectral estimate from its coefficients.

The MEM estimation (13.7.4) is a function of continuously varying frequency f. There is no special significance to specific equally spaced frequencies as there was in the FFT case. In fact, since the MEM estimate may have very sharp spectral features, one wants to be able to evaluate it on a very fine mesh near to those features, but perhaps only more coarsely farther away from them. Here is a subroutine which, given the coefficients already computed, evaluates (13.7.4) and returns the estimated power spectrum as a function of $f\Delta$ (the frequency times the sampling interval). Of course, $f\Delta$ should lie in the Nyquist range between $-1/2$ and $1/2$.

```
FUNCTION evlmem(fdt,d,m,xms)
INTEGER m
REAL evlmem,fdt,xms,d(m)
    Given d, m, xms as returned by memcof, this function returns the power spectrum estimate
    P(f) as a function of fdt = fΔ.
INTEGER i
REAL sumi,sumr
DOUBLE PRECISION theta,wi,wpi,wpr,wr,wtemp      Trigonometric recurrences in double
theta=6.28318530717959d0*fdt                        precision.
wpr=cos(theta)                          Set up for recurrence relations.
wpi=sin(theta)
wr=1.d0
wi=0.d0
sumr=1.                                 These will accumulate the denominator of (13.7.4).
```

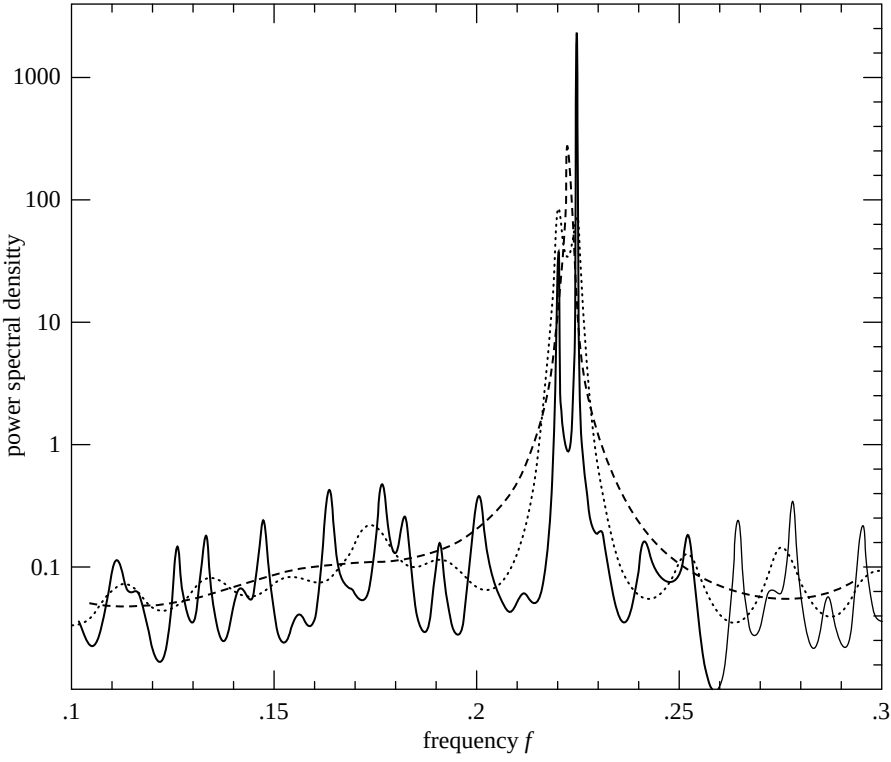


Figure 13.7.1. Sample output of maximum entropy spectral estimation. The input signal consists of 512 samples of the sum of two sinusoids of very nearly the same frequency, plus white noise with about equal power. Shown is an expanded portion of the full Nyquist frequency interval (which would extend from zero to 0.5). The dashed spectral estimate uses 20 poles; the dotted, 40; the solid, 150. With the larger number of poles, the method can resolve the distinct sinusoids; but the flat noise background is beginning to show spurious peaks. (Note logarithmic scale.)

```
sumi=0.
do 11 i=1,m                          Loop over the terms in the sum.
    wtemp=wr
    wr=wr*wpr-wi*wpi
    wi=wi*wpr+wtemp*wpi
    sumr=sumr-d(i)*sngl(wr)
    sumi=sumi-d(i)*sngl(wi)
enddo 11
evlmem=xms/(sumr**2+sumi**2)         Equation (13.7.4).
return
END
```

Be sure to evaluate $P(f)$ on a fine enough grid to *find* any narrow features that may be there! Such narrow features, if present, can contain virtually all of the power in the data. You might also wish to know how the $P(f)$ produced by the routines `memcof` and `evlmem` is normalized with respect to the mean square value of the input data vector. The answer is

$$\int_{-1/2}^{1/2} P(f\Delta)d(f\Delta) = 2\int_{0}^{1/2} P(f\Delta)d(f\Delta) = \text{mean square value of data} \qquad (13.7.8)$$

Sample spectra produced by the routines `memcof` and `evlmem` are shown in Figure 13.7.1.

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), Chapter II.

Kay, S.M., and Marple, S.L. 1981, *Proceedings of the IEEE*, vol. 69, pp. 1380–1419.

13.8 Spectral Analysis of Unevenly Sampled Data

Thus far, we have been dealing exclusively with evenly sampled data,

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{13.8.1}$$

where Δ is the sampling interval, whose reciprocal is the sampling rate. Recall also (§12.1) the significance of the Nyquist critical frequency

$$f_c \equiv \frac{1}{2\Delta} \tag{13.8.2}$$

as codified by the sampling theorem: A sampled data set like equation (13.8.1) contains *complete* information about all spectral components in a signal $h(t)$ up to the Nyquist frequency, and scrambled or *aliased* information about any signal components at frequencies larger than the Nyquist frequency. The sampling theorem thus defines both the attractiveness, and the limitation, of any analysis of an evenly spaced data set.

There are situations, however, where evenly spaced data cannot be obtained. A common case is where instrumental drop-outs occur, so that data is obtained only on a (not consecutive integer) subset of equation (13.8.1), the so-called *missing data* problem. Another case, common in observational sciences like astronomy, is that the observer cannot completely control the time of the observations, but must simply accept a certain dictated set of t_i's.

There are some obvious ways to get from unevenly spaced t_i's to evenly spaced ones, as in equation (13.8.1). Interpolation is one way: lay down a grid of evenly spaced times on your data and interpolate values onto that grid; then use FFT methods. In the missing data problem, you only have to interpolate on missing data points. If a lot of consecutive points are missing, you might as well just set them to zero, or perhaps "clamp" the value at the last measured point. However, the experience of practitioners of such interpolation techniques *is not reassuring*. Generally speaking, such techniques perform poorly. Long gaps in the data, for example, often produce a spurious bulge of power at low frequencies (wavelengths comparable to gaps).

A completely different method of spectral analysis for unevenly sampled data, one that mitigates these difficulties and has some other very desirable properties, was developed by Lomb [1], based in part on earlier work by Barning [2] and Vaníček [3], and additionally elaborated by Scargle [4]. The Lomb method (as we will call it) evaluates data, and sines and cosines, only at times t_i that are actually measured. Suppose that there are N data points $h_i \equiv h(t_i),\ i = 1, \ldots, N$. Then first find the mean and variance of the data by the usual formulas,

$$\overline{h} \equiv \frac{1}{N}\sum_{1}^{N} h_i \qquad \sigma^2 \equiv \frac{1}{N-1}\sum_{1}^{N}(h_i - \overline{h})^2 \tag{13.8.3}$$

Now, the Lomb *normalized periodogram* (spectral power as a function of angular frequency $\omega \equiv 2\pi f > 0$) is defined by

$$P_N(\omega) \equiv \frac{1}{2\sigma^2}\left\{ \frac{\left[\sum_j (h_j - \overline{h})\cos\omega(t_j - \tau)\right]^2}{\sum_j \cos^2\omega(t_j - \tau)} + \frac{\left[\sum_j (h_j - \overline{h})\sin\omega(t_j - \tau)\right]^2}{\sum_j \sin^2\omega(t_j - \tau)} \right\} \tag{13.8.4}$$

CITED REFERENCES AND FURTHER READING:

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), Chapter II.

Kay, S.M., and Marple, S.L. 1981, *Proceedings of the IEEE*, vol. 69, pp. 1380–1419.

13.8 Spectral Analysis of Unevenly Sampled Data

Thus far, we have been dealing exclusively with evenly sampled data,

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{13.8.1}$$

where Δ is the sampling interval, whose reciprocal is the sampling rate. Recall also (§12.1) the significance of the Nyquist critical frequency

$$f_c \equiv \frac{1}{2\Delta} \tag{13.8.2}$$

as codified by the sampling theorem: A sampled data set like equation (13.8.1) contains *complete* information about all spectral components in a signal $h(t)$ up to the Nyquist frequency, and scrambled or *aliased* information about any signal components at frequencies larger than the Nyquist frequency. The sampling theorem thus defines both the attractiveness, and the limitation, of any analysis of an evenly spaced data set.

There are situations, however, where evenly spaced data cannot be obtained. A common case is where instrumental drop-outs occur, so that data is obtained only on a (not consecutive integer) subset of equation (13.8.1), the so-called *missing data* problem. Another case, common in observational sciences like astronomy, is that the observer cannot completely control the time of the observations, but must simply accept a certain dictated set of t_i's.

There are some obvious ways to get from unevenly spaced t_i's to evenly spaced ones, as in equation (13.8.1). Interpolation is one way: lay down a grid of evenly spaced times on your data and interpolate values onto that grid; then use FFT methods. In the missing data problem, you only have to interpolate on missing data points. If a lot of consecutive points are missing, you might as well just set them to zero, or perhaps "clamp" the value at the last measured point. However, the experience of practitioners of such interpolation techniques *is not reassuring*. Generally speaking, such techniques perform poorly. Long gaps in the data, for example, often produce a spurious bulge of power at low frequencies (wavelengths comparable to gaps).

A completely different method of spectral analysis for unevenly sampled data, one that mitigates these difficulties and has some other very desirable properties, was developed by Lomb [1], based in part on earlier work by Barning [2] and Vaníček [3], and additionally elaborated by Scargle [4]. The Lomb method (as we will call it) evaluates data, and sines and cosines, only at times t_i that are actually measured. Suppose that there are N data points $h_i \equiv h(t_i),\ i = 1, \ldots, N$. Then first find the mean and variance of the data by the usual formulas,

$$\overline{h} \equiv \frac{1}{N}\sum_{1}^{N} h_i \qquad \sigma^2 \equiv \frac{1}{N-1}\sum_{1}^{N}(h_i - \overline{h})^2 \tag{13.8.3}$$

Now, the Lomb *normalized periodogram* (spectral power as a function of angular frequency $\omega \equiv 2\pi f > 0$) is defined by

$$P_N(\omega) \equiv \frac{1}{2\sigma^2}\left\{ \frac{\left[\sum_j (h_j - \overline{h})\cos\omega(t_j - \tau)\right]^2}{\sum_j \cos^2\omega(t_j - \tau)} + \frac{\left[\sum_j (h_j - \overline{h})\sin\omega(t_j - \tau)\right]^2}{\sum_j \sin^2\omega(t_j - \tau)} \right\} \tag{13.8.4}$$

Here τ is defined by the relation

$$\tan(2\omega\tau) = \frac{\sum_j \sin 2\omega t_j}{\sum_j \cos 2\omega t_j} \tag{13.8.5}$$

The constant τ is a kind of offset that makes $P_N(\omega)$ completely independent of shifting all the t_i's by any constant. Lomb shows that this particular choice of offset has another, deeper, effect: It makes equation (13.8.4) identical to the equation that one would obtain if one estimated the harmonic content of a data set, at a given frequency ω, by linear least-squares fitting to the model

$$h(t) = A\cos\omega t + B\sin\omega t \tag{13.8.6}$$

This fact gives some insight into why the method can give results superior to FFT methods: It weights the data on a "per point" basis instead of on a "per time interval" basis, when uneven sampling can render the latter seriously in error.

A very common occurrence is that the measured data points h_i are the sum of a periodic signal and independent (white) Gaussian noise. If we are trying to determine the presence or absence of such a periodic signal, we want to be able to give a quantitative answer to the question, "How significant is a peak in the spectrum $P_N(\omega)$?" In this question, the null hypothesis is that the data values are independent Gaussian random values. A very nice property of the Lomb normalized periodogram is that the viability of the null hypothesis can be tested fairly rigorously, as we now discuss.

The word "normalized" refers to the factor σ^2 in the denominator of equation (13.8.4). Scargle [4] shows that with this normalization, at any particular ω and *in the case of the null hypothesis*, $P_N(\omega)$ has an exponential probability distribution with unit mean. In other words, the probability that $P_N(\omega)$ will be between some positive z and $z + dz$ is $\exp(-z)dz$. It readily follows that, if we scan some M *independent* frequencies, the probability that none give values larger than z is $(1 - e^{-z})^M$. So

$$P(> z) \equiv 1 - (1 - e^{-z})^M \tag{13.8.7}$$

is the false-alarm probability of the null hypothesis, that is, the *significance level* of any peak in $P_N(\omega)$ that we do see. A small value for the false-alarm probability indicates a highly significant periodic signal.

To evaluate this significance, we need to know M. After all, the more frequencies we look at, the less significant is some one modest bump in the spectrum. (Look long enough, find anything!) A typical procedure will be to plot $P_N(\omega)$ as a function of many closely spaced frequencies in some large frequency range. How many of these are independent?

Before answering, let us first see how accurately we need to know M. The interesting region is where the significance is a small (significant) number, $\ll 1$. There, equation (13.8.7) can be series expanded to give

$$P(> z) \approx Me^{-z} \tag{13.8.8}$$

We see that the significance scales linearly with M. Practical significance levels are numbers like 0.05, 0.01, 0.001, etc. An error of even $\pm 50\%$ in the estimated significance is often tolerable, since quoted significance levels are typically spaced apart by factors of 5 or 10. So our estimate of M need not be very accurate.

Horne and Baliunas [5] give results from extensive Monte Carlo experiments for determining M in various cases. In general M depends on the number of frequencies sampled, the number of data points N, and their detailed spacing. It turns out that M is very nearly equal to N when the data points are approximately equally spaced, and when the sampled frequencies "fill" (oversample) the frequency range from 0 to the Nyquist frequency f_c (equation 13.8.2). Further, the value of M is not importantly different for random spacing of the data points than for equal spacing. When a larger frequency range than the Nyquist range is sampled, M increases proportionally. About the only case where M differs significantly from the case of evenly spaced points is when the points are closely clumped, say into groups of 3; then (as one would expect) the number of independent frequencies is reduced by a factor of about 3.

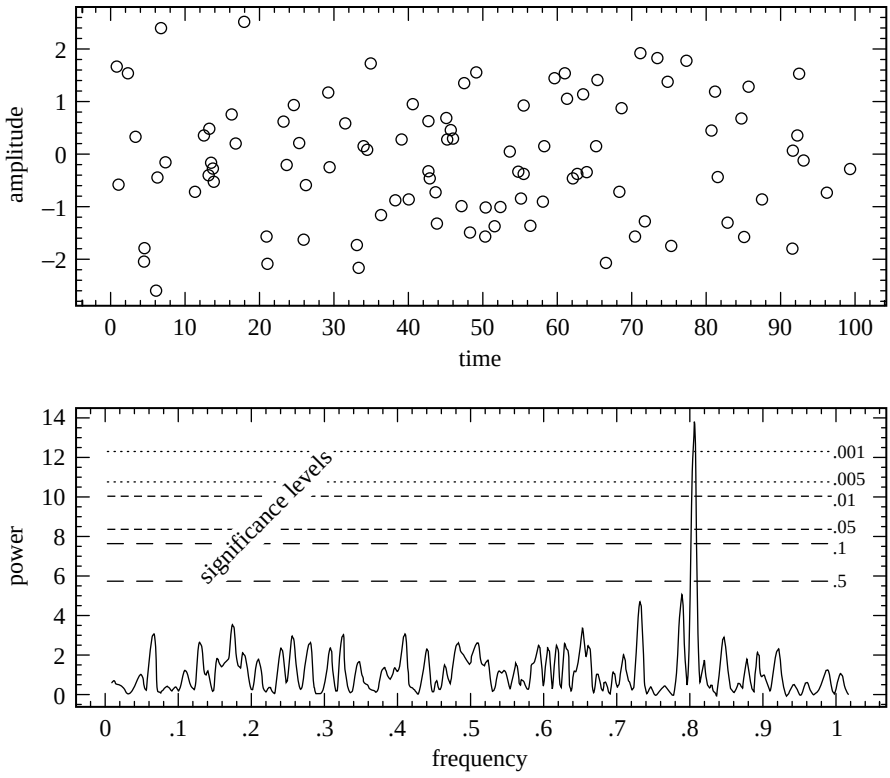


Figure 13.8.1. Example of the Lomb algorithm in action. The 100 data points (upper figure) are at random times between 0 and 100. Their sinusoidal component is readily uncovered (lower figure) by the algorithm, at a significance level better than 0.001. If the 100 data points had been evenly spaced at unit interval, the Nyquist critical frequency would have been 0.5. Note that, for these unevenly spaced points, there is no visible aliasing into the Nyquist range.

The program `period`, below, calculates an effective value for M based on the above rough-and-ready rules and assumes that there is no important clumping. This will be adequate for most purposes. In any particular case, if it really matters, it is not too difficult to compute a better value of M by simple Monte Carlo: Holding fixed the number of data points and their locations t_i, generate synthetic data sets of Gaussian (normal) deviates, find the largest values of $P_N(\omega)$ for each such data set (using the accompanying program), and fit the resulting distribution for M in equation (13.8.7).

Figure 13.8.1 shows the results of applying the method as discussed so far. In the upper figure, the data points are plotted against time. Their number is $N = 100$, and their distribution in t is Poisson random. There is certainly no sinusoidal signal evident to the eye. The lower figure plots $P_N(\omega)$ against frequency $f = \omega/2\pi$. The Nyquist critical frequency that would obtain if the points were evenly spaced is at $f = f_c = 0.5$. Since we have searched up to about twice that frequency, and oversampled the f's to the point where successive values of $P_N(\omega)$ vary smoothly, we take $M = 2N$. The horizontal dashed and dotted lines are (respectively from bottom to top) significance levels 0.5, 0.1, 0.05, 0.01, 0.005, and 0.001. One sees a highly significant peak at a frequency of 0.81. That is in fact the frequency of the sine wave that is present in the data. (You will have to take our word for this!)

Note that two other peaks approach, but do not exceed the 50% significance level; that is about what one might expect by chance. It is also worth commenting on the fact that the significant peak was found (correctly) *above the Nyquist frequency* and without any significant aliasing down into the Nyquist interval! That would not be possible for evenly spaced data. It is possible here because the randomly spaced data has *some* points spaced much closer than

the "average" sampling rate, and these remove ambiguity from any aliasing.

Implementation of the normalized periodogram in code is straightforward, with, however, a few points to be kept in mind. We are dealing with a *slow* algorithm. Typically, for N data points, we may wish to examine on the order of $2N$ or $4N$ frequencies. Each combination of frequency and data point has, in equations (13.8.4) and (13.8.5), not just a few adds or multiplies, but four calls to trigonometric functions; the operations count can easily reach several hundred times N^2. It is highly desirable — in fact results in a factor 4 speedup — to replace these trigonometric calls by recurrences. That is possible only if the sequence of frequencies examined is a linear sequence. Since such a sequence is probably what most users would want anyway, we have built this into the implementation.

At the end of this section we describe a way to evaluate equations (13.8.4) and (13.8.5) — approximately, but to any desired degree of approximation — by a fast method [6] whose operation count goes only as $N \log N$. This faster method should be used for long data sets.

The lowest independent frequency f to be examined is the inverse of the span of the input data, $\max_i(t_i) - \min_i(t_i) \equiv T$. This is the frequency such that the data can include one complete cycle. In subtracting off the data's mean, equation (13.8.4) already assumed that you are not interested in the data's zero-frequency piece — which is just that mean value. In an FFT method, higher independent frequencies would be integer multiples of $1/T$. Because we are interested in the statistical significance of any peak that may occur, however, we had better (over-) sample more finely than at interval $1/T$, so that sample points lie close to the top of any peak. Thus, the accompanying program includes an oversampling parameter, called `ofac`; a value `ofac` $\gtrsim 4$ might be typical in use. We also want to specify how high in frequency to go, say f_{hi}. One guide to choosing f_{hi} is to compare it with the Nyquist frequency f_c which would obtain if the N data points were evenly spaced over the same span T, that is $f_c = N/(2T)$. The accompanying program includes an input parameter `hifac`, defined as f_{hi}/f_c. The number of different frequencies N_P returned by the program is then given by

$$N_P = \frac{\texttt{ofac} \times \texttt{hifac}}{2} N \tag{13.8.9}$$

(You have to remember to dimension the output arrays to at least this size.)

The code does the trigonometric recurrences in double precision and embodies a few tricks with trigonometric identities, to decrease roundoff errors. If you are an aficionado of such things you can puzzle it out. A final detail is that equation (13.8.7) will fail because of roundoff error if z is too large; but equation (13.8.8) is fine in this regime.

```
      SUBROUTINE period(x,y,n,ofac,hifac,px,py,np,nout,jmax,prob)
      INTEGER jmax,n,nout,np,NMAX
      REAL hifac,ofac,prob,px(np),py(np),x(n),y(n)
      PARAMETER (NMAX=2000)                    Maximum expected value of n.
C     USES avevar
        Given n data points with abscissas x(1:n) (which need not be equally spaced) and ordinates
        y(1:n), and given a desired oversampling factor ofac (a typical value being 4 or larger),
        this routine fills array px with an increasing sequence of frequencies (not angular frequencies)
        up to hifac times the "average" Nyquist frequency, and fills array py with the values of
        the Lomb normalized periodogram at those frequencies. The arrays x and y are not altered.
        np, the dimension of px and py, must be large enough to contain the output, or an error
        (pause) results. The routine also returns jmax such that py(jmax) is the maximum element
        in py, and prob, an estimate of the significance of that maximum against the hypothesis of
        random noise. A small value of prob indicates that a significant periodic signal is present.
      INTEGER i,j
      REAL ave,c,cc,cwtau,effm,expy,pnow,pymax,s,ss,sumc,sumcy,
     *     sums,sumsh,sumsy,swtau,var,wtau,xave,xdif,xmax,xmin,yy
      DOUBLE PRECISION arg,wtemp,wi(NMAX),wpi(NMAX),
     *     wpr(NMAX),wr(NMAX),TWOPID
      PARAMETER (TWOPID=6.2831853071795865D0)
      nout=0.5*ofac*hifac*n
      if(nout.gt.np) pause 'output arrays too short in period'
      call avevar(y,n,ave,var)      Get mean and variance of the input data.
      xmax=x(1)
      xmin=x(1)                     Go through data to get the range of abscissas.
```

```
do 11 j=1,n
    if(x(j).gt.xmax)xmax=x(j)
    if(x(j).lt.xmin)xmin=x(j)
enddo 11
xdif=xmax-xmin
xave=0.5*(xmax+xmin)
pymax=0.
pnow=1./(xdif*ofac)                    Starting frequency.
do 12 j=1,n                            Initialize values for the trigonometric recurrences
    arg=TWOPID*((x(j)-xave)*pnow)          at each data point. The recurrences are done
    wpr(j)=-2.d0*sin(0.5d0*arg)**2         in double precision.
    wpi(j)=sin(arg)
    wr(j)=cos(arg)
    wi(j)=wpi(j)
enddo 12
do 15 i=1,nout                 Main loop over the frequencies to be evaluated.
    px(i)=pnow
    sumsh=0.
    sumc=0.                    First, loop over the data to get τ and related quantities.
    do 13 j=1,n
        c=wr(j)
        s=wi(j)
        sumsh=sumsh+s*c
        sumc=sumc+(c-s)*(c+s)
    enddo 13
    wtau=0.5*atan2(2.*sumsh,sumc)
    swtau=sin(wtau)
    cwtau=cos(wtau)
    sums=0.
    sumc=0.
    sumsy=0.                   Then, loop over the data again to get the periodogram value.
    sumcy=0.
    do 14 j=1,n
        s=wi(j)
        c=wr(j)
        ss=s*cwtau-c*swtau
        cc=c*cwtau+s*swtau
        sums=sums+ss**2
        sumc=sumc+cc**2
        yy=y(j)-ave
        sumsy=sumsy+yy*ss
        sumcy=sumcy+yy*cc
        wtemp=wr(j)            Update the trigonometric recurrences.
        wr(j)=(wr(j)*wpr(j)-wi(j)*wpi(j))+wr(j)
        wi(j)=(wi(j)*wpr(j)+wtemp*wpi(j))+wi(j)
    enddo 14
    py(i)=0.5*(sumcy**2/sumc+sumsy**2/sums)/var
    if (py(i).ge.pymax) then
        pymax=py(i)
        jmax=i
    endif
    pnow=pnow+1./(ofac*xdif) The next frequency.
enddo 15
expy=exp(-pymax)               Evaluate statistical significance of the maximum.
effm=2.*nout/ofac
prob=effm*expy
if(prob.gt.0.01)prob=1.-(1.-expy)**effm
return
END
```

Fast Computation of the Lomb Periodogram

We here show how equations (13.8.4) and (13.8.5) can be calculated — approximately, but to any desired precision — with an operation count only of order $N_P \log N_P$. The method uses the FFT, but it is in no sense an FFT periodogram of the data. It is an actual evaluation of equations (13.8.4) and (13.8.5), the Lomb normalized periodogram, with exactly that method's strengths and weaknesses. This fast algorithm, due to Press and Rybicki [6], makes feasible the application of the Lomb method to data sets at least as large as 10^6 points; it is already faster than straightforward evaluation of equations (13.8.4) and (13.8.5) for data sets as small as 60 or 100 points.

Notice that the trigonometric sums that occur in equations (13.8.5) and (13.8.4) can be reduced to four simpler sums. If we define

$$S_h \equiv \sum_{j=1}^{N} (h_j - \bar{h}) \sin(\omega t_j) \qquad C_h \equiv \sum_{j=1}^{N} (h_j - \bar{h}) \cos(\omega t_j) \tag{13.8.10}$$

and

$$S_2 \equiv \sum_{j=1}^{N} \sin(2\omega t_j) \qquad C_2 \equiv \sum_{j=1}^{N} \cos(2\omega t_j) \tag{13.8.11}$$

then

$$\begin{aligned}
\sum_{j=1}^{N} (h_j - \bar{h}) \cos\omega(t_j - \tau) &= C_h \cos\omega\tau + S_h \sin\omega\tau \\
\sum_{j=1}^{N} (h_j - \bar{h}) \sin\omega(t_j - \tau) &= S_h \cos\omega\tau - C_h \sin\omega\tau \\
\sum_{j=1}^{N} \cos^2\omega(t_j - \tau) &= \frac{N}{2} + \frac{1}{2} C_2 \cos(2\omega\tau) + \frac{1}{2} S_2 \sin(2\omega\tau) \\
\sum_{j=1}^{N} \sin^2\omega(t_j - \tau) &= \frac{N}{2} - \frac{1}{2} C_2 \cos(2\omega\tau) - \frac{1}{2} S_2 \sin(2\omega\tau)
\end{aligned} \tag{13.8.12}$$

Now notice that *if* the t_js *were* evenly spaced, then the four quantities S_h, C_h, S_2, and C_2 could be evaluated by two complex FFTs, and the results could then be substituted back through equation (13.8.12) to evaluate equations (13.8.5) and (13.8.4). The problem is therefore only to evaluate equations (13.8.10) and (13.8.11) for unevenly spaced data.

Interpolation, or rather reverse interpolation — we will here call it *extirpolation* — provides the key. Interpolation, as classically understood, uses several function values on a regular mesh to construct an accurate approximation at an arbitrary point. Extirpolation, just the opposite, *replaces* a function value at an arbitrary point by several function values on a regular mesh, doing this in such a way that sums over the mesh are an accurate approximation to sums over the original arbitrary point.

It is not hard to see that the weight functions for extirpolation are identical to those for interpolation. Suppose that the function $h(t)$ to be extirpolated is known only at the discrete (unevenly spaced) points $h(t_i) \equiv h_i$, and that the function $g(t)$ (which will be, e.g., $\cos\omega t$) can be evaluated anywhere. Let $\hat{t}_k$ be a sequence of evenly spaced points on a regular mesh. Then Lagrange interpolation (§3.1) gives an approximation of the form

$$g(t) \approx \sum_k w_k(t) g(\hat{t}_k) \tag{13.8.13}$$

where $w_k(t)$ are interpolation weights. Now let us evaluate a sum of interest by the following scheme:

$$\sum_{j=1}^{N} h_j g(t_j) \approx \sum_{j=1}^{N} h_j \left[\sum_k w_k(t_j) g(\hat{t}_k) \right] = \sum_k \left[\sum_{j=1}^{N} h_j w_k(t_j) \right] g(\hat{t}_k) \equiv \sum_k \widehat{h}_k \, g(\hat{t}_k) \tag{13.8.14}$$

Here $\widehat{h}_k \equiv \sum_j h_j w_k(t_j)$. Notice that equation (13.8.14) replaces the original sum by one on the regular mesh. Notice also that the accuracy of equation (13.8.13) depends only on the fineness of the mesh with respect to the function g and has nothing to do with the spacing of the points t_j or the function h; therefore the accuracy of equation (13.8.14) also has this property.

The general outline of the fast evaluation method is therefore this: (i) Choose a mesh size large enough to accommodate some desired oversampling factor, and large enough to have several extirpolation points per half-wavelength of the highest frequency of interest. (ii) Extirpolate the values h_i onto the mesh and take the FFT; this gives S_h and C_h in equation (13.8.10). (iii) Extirpolate the constant values 1 onto another mesh, and take its FFT; this, with some manipulation, gives S_2 and C_2 in equation (13.8.11). (iv) Evaluate equations (13.8.12), (13.8.5), and (13.8.4), in that order.

There are several other tricks involved in implementing this algorithm efficiently. You can figure most out from the code, but we will mention the following points: (a) A nice way to get transform values at frequencies 2ω instead of ω is to stretch the time-domain data by a factor 2, and then wrap it to double-cover the original length. (This trick goes back to Tukey.) In the program, this appears as a modulo function. (b) Trigonometric identities are used to get from the left-hand side of equation (13.8.5) to the various needed trigonometric functions of $\omega\tau$. `FORTRAN` identifiers like (e.g.) `cwt` and `hs2wt` represent quantities like (e.g.) $\cos\omega\tau$ and $\frac{1}{2}\sin(2\omega\tau)$. (c) The subroutine `spread` does extirpolation onto the M most nearly centered mesh points around an arbitrary point; its turgid code evaluates coefficients of the Lagrange interpolating polynomials, in an efficient manner.

```
      SUBROUTINE fasper(x,y,n,ofac,hifac,wk1,wk2,nwk,nout,jmax,prob)
      INTEGER jmax,n,nout,nwk,MACC
      REAL hifac,ofac,prob,wk1(nwk),wk2(nwk),x(n),y(n)
      PARAMETER (MACC=4)              Number of interpolation points per 1/4 cycle of highest fre-
C     USES avevar,realft,spread       quency.
        Given n data points with abscissas x (which need not be equally spaced) and ordinates y,
        and given a desired oversampling factor ofac (a typical value being 4 or larger), this routine
        fills array wk1 with a sequence of nout increasing frequencies (not angular frequencies) up
        to hifac times the "average" Nyquist frequency, and fills array wk2 with the values of the
        Lomb normalized periodogram at those frequencies. The arrays x and y are not altered.
        nwk, the dimension of wk1 and wk2, must be large enough for intermediate work space,
        or an error (pause) results. The routine also returns jmax such that wk2(jmax) is the
        maximum element in wk2, and prob, an estimate of the significance of that maximum
        against the hypothesis of random noise. A small value of prob indicates that a significant
        periodic signal is present.
      INTEGER j,k,ndim,nfreq,nfreqt
      REAL ave,ck,ckk,cterm,cwt,den,df,effm,expy,fac,fndim,hc2wt,
     *     hs2wt,hypo,pmax,sterm,swt,var,xdif,xmax,xmin
      EXTERNAL spread
      nout=0.5*ofac*hifac*n
      nfreqt=ofac*hifac*n*MACC        Size the FFT as next power of 2 above nfreqt.
      nfreq=64
1     if (nfreq.lt.nfreqt) then
        nfreq=nfreq*2
      goto 1
      endif
      ndim=2*nfreq
      if(ndim.gt.nwk) pause 'workspaces too small in fasper'
      call avevar(y,n,ave,var)        Compute the mean, variance, and range of the data.
      xmin=x(1)
      xmax=xmin
      do 11 j=2,n
        if(x(j).lt.xmin)xmin=x(j)
        if(x(j).gt.xmax)xmax=x(j)
      enddo 11
      xdif=xmax-xmin
      do 12 j=1,ndim                  Zero the workspaces.
        wk1(j)=0.
        wk2(j)=0.
      enddo 12
```

```
fac=ndim/(xdif*ofac)
fndim=ndim
do 13 j=1,n                    Extirpolate the data into the workspaces.
    ck=1.+mod((x(j)-xmin)*fac,fndim)
    ckk=1.+mod(2.*(ck-1.),fndim)
    call spread(y(j)-ave,wk1,ndim,ck,MACC)
    call spread(1.,wk2,ndim,ckk,MACC)
enddo 13
call realft(wk1,ndim,1)        Take the Fast Fourier Transforms.
call realft(wk2,ndim,1)
df=1./(xdif*ofac)
k=3
pmax=-1.
do 14 j=1,nout                 Compute the Lomb value for each frequency.
    hypo=sqrt(wk2(k)**2+wk2(k+1)**2)
    hc2wt=0.5*wk2(k)/hypo
    hs2wt=0.5*wk2(k+1)/hypo
    cwt=sqrt(0.5+hc2wt)
    swt=sign(sqrt(0.5-hc2wt),hs2wt)
    den=0.5*n+hc2wt*wk2(k)+hs2wt*wk2(k+1)
    cterm=(cwt*wk1(k)+swt*wk1(k+1))**2/den
    sterm=(cwt*wk1(k+1)-swt*wk1(k))**2/(n-den)
    wk1(j)=j*df
    wk2(j)=(cterm+sterm)/(2.*var)
    if (wk2(j).gt.pmax) then
        pmax=wk2(j)
        jmax=j
    endif
    k=k+2
enddo 14                       Estimate significance of largest peak value.
expy=exp(-pmax)
effm=2.*nout/ofac
prob=effm*expy
if(prob.gt.0.01)prob=1.-(1.-expy)**effm
return
END
```

```
SUBROUTINE spread(y,yy,n,x,m)
INTEGER m,n
REAL x,y,yy(n)
    Given an array yy of length n, extirpolate (spread) a value y into m actual array elements
    that best approximate the "fictional" (i.e., possibly noninteger) array element number x.
    The weights used are coefficients of the Lagrange interpolating polynomial.
INTEGER ihi,ilo,ix,j,nden,nfac(10)
REAL fac
SAVE nfac
DATA nfac /1,1,2,6,24,120,720,5040,40320,362880/
if(m.gt.10) pause 'factorial table too small in spread'
ix=x
if(x.eq.float(ix))then
    yy(ix)=yy(ix)+y
else
    ilo=min(max(int(x-0.5*m+1.0),1),n-m+1)
    ihi=ilo+m-1
    nden=nfac(m)
    fac=x-ilo
    do 11 j=ilo+1,ihi
        fac=fac*(x-j)
    enddo 11
    yy(ihi)=yy(ihi)+y*fac/(nden*(x-ihi))
    do 12 j=ihi-1,ilo,-1
        nden=(nden/(j+1-ilo))*(j-ihi)
```

```
            yy(j)=yy(j)+y*fac/(nden*(x-j))
        enddo 12
    endif
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Lomb, N.R. 1976, *Astrophysics and Space Science*, vol. 39, pp. 447–462. [1]

Barning, F.J.M. 1963, *Bulletin of the Astronomical Institutes of the Netherlands*, vol. 17, pp. 22–28. [2]

Vaníček, P. 1971, *Astrophysics and Space Science*, vol. 12, pp. 10–33. [3]

Scargle, J.D. 1982, *Astrophysical Journal*, vol. 263, pp. 835–853. [4]

Horne, J.H., and Baliunas, S.L. 1986, *Astrophysical Journal*, vol. 302, pp. 757–763. [5]

Press, W.H. and Rybicki, G.B. 1989, *Astrophysical Journal*, vol. 338, pp. 277–280. [6]

13.9 Computing Fourier Integrals Using the FFT

Not uncommonly, one wants to calculate accurate numerical values for integrals of the form

$$I = \int_a^b e^{i\omega t} h(t) dt \ , \tag{13.9.1}$$

or the equivalent real and imaginary parts

$$I_c = \int_a^b \cos(\omega t) h(t) dt \qquad I_s = \int_a^b \sin(\omega t) h(t) dt \ , \tag{13.9.2}$$

and one wants to evaluate this integral for many different values of ω. In cases of interest, $h(t)$ is often a smooth function, but it is not necessarily periodic in $[a, b]$, nor does it necessarily go to zero at a or b. While it seems intuitively obvious that the *force majeure* of the FFT ought to be applicable to this problem, doing so turns out to be a surprisingly subtle matter, as we will now see.

Let us first approach the problem naively, to see where the difficulty lies. Divide the interval $[a, b]$ into M subintervals, where M is a large integer, and define

$$\Delta \equiv \frac{b-a}{M} \ , \quad t_j \equiv a + j\Delta \ , \quad h_j \equiv h(t_j) \ , \quad j = 0, \ldots, M \tag{13.9.3}$$

Notice that $h_0 = h(a)$ and $h_M = h(b)$, and that there are $M + 1$ values h_j. We can approximate the integral I by a sum,

$$I \approx \Delta \sum_{j=0}^{M-1} h_j \exp(i\omega t_j) \tag{13.9.4}$$

which is at any rate first-order accurate. (If we centered the h_j's and the t_j's in the intervals, we could be accurate to second order.) Now for certain values of ω and M, the sum in equation (13.9.4) can be made into a discrete Fourier transform, or DFT, and evaluated by the fast Fourier transform (FFT) algorithm. In particular, we can choose M to be an integer power of 2, and define a set of special ω's by

$$\omega_m \Delta \equiv \frac{2\pi m}{M} \tag{13.9.5}$$

```
            yy(j)=yy(j)+y*fac/(nden*(x-j))
        enddo 12
    endif
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Lomb, N.R. 1976, *Astrophysics and Space Science*, vol. 39, pp. 447–462. [1]

Barning, F.J.M. 1963, *Bulletin of the Astronomical Institutes of the Netherlands*, vol. 17, pp. 22–28. [2]

Vaníček, P. 1971, *Astrophysics and Space Science*, vol. 12, pp. 10–33. [3]

Scargle, J.D. 1982, *Astrophysical Journal*, vol. 263, pp. 835–853. [4]

Horne, J.H., and Baliunas, S.L. 1986, *Astrophysical Journal*, vol. 302, pp. 757–763. [5]

Press, W.H. and Rybicki, G.B. 1989, *Astrophysical Journal*, vol. 338, pp. 277–280. [6]

13.9 Computing Fourier Integrals Using the FFT

Not uncommonly, one wants to calculate accurate numerical values for integrals of the form

$$I = \int_a^b e^{i\omega t} h(t) dt \ , \tag{13.9.1}$$

or the equivalent real and imaginary parts

$$I_c = \int_a^b \cos(\omega t) h(t) dt \qquad I_s = \int_a^b \sin(\omega t) h(t) dt \ , \tag{13.9.2}$$

and one wants to evaluate this integral for many different values of ω. In cases of interest, $h(t)$ is often a smooth function, but it is not necessarily periodic in $[a, b]$, nor does it necessarily go to zero at a or b. While it seems intuitively obvious that the *force majeure* of the FFT ought to be applicable to this problem, doing so turns out to be a surprisingly subtle matter, as we will now see.

Let us first approach the problem naively, to see where the difficulty lies. Divide the interval $[a, b]$ into M subintervals, where M is a large integer, and define

$$\Delta \equiv \frac{b-a}{M} \ , \quad t_j \equiv a + j\Delta \ , \quad h_j \equiv h(t_j) \ , \quad j = 0, \ldots, M \tag{13.9.3}$$

Notice that $h_0 = h(a)$ and $h_M = h(b)$, and that there are $M + 1$ values h_j. We can approximate the integral I by a sum,

$$I \approx \Delta \sum_{j=0}^{M-1} h_j \exp(i\omega t_j) \tag{13.9.4}$$

which is at any rate first-order accurate. (If we centered the h_j's and the t_j's in the intervals, we could be accurate to second order.) Now for certain values of ω and M, the sum in equation (13.9.4) can be made into a discrete Fourier transform, or DFT, and evaluated by the fast Fourier transform (FFT) algorithm. In particular, we can choose M to be an integer power of 2, and define a set of special ω's by

$$\omega_m \Delta \equiv \frac{2\pi m}{M} \tag{13.9.5}$$

where m has the values $m = 0, 1, \ldots, M/2 - 1$. Then equation (13.9.4) becomes

$$I(\omega_m) \approx \Delta e^{i\omega_m a} \sum_{j=0}^{M-1} h_j e^{2\pi i m j/M} = \Delta e^{i\omega_m a} [\mathrm{DFT}(h_0 \ldots h_{M-1})]_m \tag{13.9.6}$$

Equation (13.9.6), while simple and clear, is emphatically *not recommended* for use: It is likely to give wrong answers!

The problem lies in the oscillatory nature of the integral (13.9.1). If $h(t)$ is at all smooth, and if ω is large enough to imply several cycles in the interval $[a, b]$ — in fact, ω_m in equation (13.9.5) gives exactly m cycles — then the value of I is typically very small, so small that it is easily swamped by first-order, or even (with centered values) second-order, truncation error. Furthermore, the characteristic "small parameter" that occurs in the error term is not $\Delta/(b-a) = 1/M$, as it would be if the integrand were not oscillatory, but $\omega\Delta$, which can be as large as π for ω's within the Nyquist interval of the DFT (cf. equation 13.9.5). The result is that equation (13.9.6) becomes systematically inaccurate as ω increases.

It is a sobering exercise to implement equation (13.9.6) for an integral that can be done analytically, and to see just how bad it is. We recommend that you try it.

Let us therefore turn to a more sophisticated treatment. Given the sampled points h_j, we can approximate the function $h(t)$ everywhere in the interval $[a, b]$ by interpolation on nearby h_j's. The simplest case is linear interpolation, using the two nearest h_j's, one to the left and one to the right. A higher-order interpolation, e.g., would be cubic interpolation, using two points to the left and two to the right — except in the first and last subintervals, where we must interpolate with three h_j's on one side, one on the other.

The formulas for such interpolation schemes are (piecewise) polynomial in the independent variable t, but with coefficients that are of course linear in the function values h_j. Although one does not usually think of it in this way, interpolation can be viewed as approximating a function by a sum of kernel functions (which depend only on the interpolation scheme) times sample values (which depend only on the function). Let us write

$$h(t) \approx \sum_{j=0}^{M} h_j \, \psi\left(\frac{t - t_j}{\Delta}\right) + \sum_{j=\mathrm{endpoints}} h_j \, \varphi_j\left(\frac{t - t_j}{\Delta}\right) \tag{13.9.7}$$

Here $\psi(s)$ is the kernel function of an interior point: It is zero for s sufficiently negative or sufficiently positive, and becomes nonzero only when s is in the range where the h_j multiplying it is actually used in the interpolation. We always have $\psi(0) = 1$ and $\psi(m) = 0$, $m = \pm 1, \pm 2, \ldots$, since interpolation right on a sample point should give the sampled function value. For linear interpolation $\psi(s)$ is piecewise linear, rises from 0 to 1 for s in $(-1, 0)$, and falls back to 0 for s in $(0, 1)$. For higher-order interpolation, $\psi(s)$ is made up piecewise of segments of Lagrange interpolation polynomials. It has discontinuous derivatives at integer values of s, where the pieces join, because the set of points used in the interpolation changes discretely.

As already remarked, the subintervals closest to a and b require different (noncentered) interpolation formulas. This is reflected in equation (13.9.7) by the second sum, with the special endpoint kernels $\varphi_j(s)$. Actually, for reasons that will become clearer below, we have included *all* the points in the *first* sum (with kernel ψ), so the φ_j's are actually differences between true endpoint kernels and the interior kernel ψ. It is a tedious, but straightforward, exercise to write down all the $\varphi_j(s)$'s for any particular order of interpolation, each one consisting of differences of Lagrange interpolating polynomials spliced together piecewise.

Now apply the integral operator $\int_a^b dt \exp(i\omega t)$ to both sides of equation (13.9.7), interchange the sums and integral, and make the changes of variable $s = (t - t_j)/\Delta$ in the first sum, $s = (t - a)/\Delta$ in the second sum. The result is

$$I \approx \Delta e^{i\omega a} \left[W(\theta) \sum_{j=0}^{M} h_j e^{ij\theta} + \sum_{j=\mathrm{endpoints}} h_j \alpha_j(\theta) \right] \tag{13.9.8}$$

Here $\theta \equiv \omega\Delta$, and the functions $W(\theta)$ and $\alpha_j(\theta)$ are defined by

$$W(\theta) \equiv \int_{-\infty}^{\infty} ds \, e^{i\theta s} \psi(s) \tag{13.9.9}$$

$$\alpha_j(\theta) \equiv \int_{-\infty}^{\infty} ds\, e^{i\theta s} \varphi_j(s-j) \tag{13.9.10}$$

The key point is that equations (13.9.9) and (13.9.10) can be evaluated, analytically, once and for all, for any given interpolation scheme. Then equation (13.9.8) is an algorithm for applying "endpoint corrections" to a sum which (as we will see) can be done using the FFT, giving a result with high-order accuracy.

We will consider only interpolations that are left-right symmetric. Then symmetry implies

$$\varphi_{M-j}(s) = \varphi_j(-s) \qquad \alpha_{M-j}(\theta) = e^{i\theta M}\alpha_j^*(\theta) = e^{i\omega(b-a)}\alpha_j^*(\theta) \tag{13.9.11}$$

where * denotes complex conjugation. Also, $\psi(s) = \psi(-s)$ implies that $W(\theta)$ is real.

Turn now to the first sum in equation (13.9.8), which we want to do by FFT methods. To do so, choose some N that is an integer power of 2 with $N \geq M+1$. (Note that M need not be a power of two, so $M = N-1$ is allowed.) If $N > M+1$, define $h_j \equiv 0$, $M+1 < j \leq N-1$, i.e., "zero pad" the array of h_j's so that j takes on the range $0 \leq j \leq N-1$. Then the sum can be done as a DFT for the special values $\omega = \omega_n$ given by

$$\omega_n \Delta \equiv \frac{2\pi n}{N} \equiv \theta \qquad n = 0, 1, \ldots, \frac{N}{2} - 1 \tag{13.9.12}$$

For fixed M, the larger N is chosen, the finer the sampling in frequency space. The value M, on the other hand, determines the *highest* frequency sampled, since Δ decreases with increasing M (equation 13.9.3), and the largest value of $\omega\Delta$ is always just under π (equation 13.9.12). In general it is advantageous to oversample by *at least* a factor of 4, i.e., $N > 4M$ (see below). We can now rewrite equation (13.9.8) in its final form as

$$\begin{aligned} I(\omega_n) = \Delta e^{i\omega_n a} \Big\{ & W(\theta)[\mathrm{DFT}(h_0 \ldots h_{N-1})]_n \\ & + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 + \ldots \\ & + e^{i\omega(b-a)} \left[\alpha_0^*(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} + \alpha_3^*(\theta)h_{M-3} + \ldots \right] \Big\} \end{aligned} \tag{13.9.13}$$

For cubic (or lower) polynomial interpolation, at most the terms explicitly shown above are nonzero; the ellipses (. . .) can therefore be ignored, and we need explicit forms only for the functions $W, \alpha_0, \alpha_1, \alpha_2, \alpha_3$, calculated with equations (13.9.9) and (13.9.10). We have worked these out for you, in the trapezoidal (second-order) and cubic (fourth-order) cases. Here are the results, along with the first few terms of their power series expansions for small θ:

Trapezoidal order:

$$W(\theta) = \frac{2(1-\cos\theta)}{\theta^2} \approx 1 - \frac{1}{12}\theta^2 + \frac{1}{360}\theta^4 - \frac{1}{20160}\theta^6$$

$$\begin{aligned} \alpha_0(\theta) &= -\frac{(1-\cos\theta)}{\theta^2} + i\frac{(\theta - \sin\theta)}{\theta^2} \\ &\approx -\frac{1}{2} + \frac{1}{24}\theta^2 - \frac{1}{720}\theta^4 + \frac{1}{40320}\theta^6 + i\theta\left(\frac{1}{6} - \frac{1}{120}\theta^2 + \frac{1}{5040}\theta^4 - \frac{1}{362880}\theta^6\right) \end{aligned}$$

$$\alpha_1 = \alpha_2 = \alpha_3 = 0$$

Cubic order:

$$W(\theta) = \left(\frac{6+\theta^2}{3\theta^4}\right)(3 - 4\cos\theta + \cos 2\theta) \approx 1 - \frac{11}{720}\theta^4 + \frac{23}{15120}\theta^6$$

$$\begin{aligned}\alpha_0(\theta) &= \frac{(-42+5\theta^2)+(6+\theta^2)(8\cos\theta-\cos 2\theta)}{6\theta^4} + i\frac{(-12\theta+6\theta^3)+(6+\theta^2)\sin 2\theta}{6\theta^4} \\ &\approx -\frac{2}{3} + \frac{1}{45}\theta^2 + \frac{103}{15120}\theta^4 - \frac{169}{226800}\theta^6 + i\theta\left(\frac{2}{45} + \frac{2}{105}\theta^2 - \frac{8}{2835}\theta^4 + \frac{86}{467775}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_1(\theta) &= \frac{14(3-\theta^2) - 7(6+\theta^2)\cos\theta}{6\theta^4} + i\frac{30\theta - 5(6+\theta^2)\sin\theta}{6\theta^4} \\ &\approx \frac{7}{24} - \frac{7}{180}\theta^2 + \frac{5}{3456}\theta^4 - \frac{7}{259200}\theta^6 + i\theta\left(\frac{7}{72} - \frac{1}{168}\theta^2 + \frac{11}{72576}\theta^4 - \frac{13}{5987520}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_2(\theta) &= \frac{-4(3-\theta^2) + 2(6+\theta^2)\cos\theta}{3\theta^4} + i\frac{-12\theta + 2(6+\theta^2)\sin\theta}{3\theta^4} \\ &\approx -\frac{1}{6} + \frac{1}{45}\theta^2 - \frac{5}{6048}\theta^4 + \frac{1}{64800}\theta^6 + i\theta\left(-\frac{7}{90} + \frac{1}{210}\theta^2 - \frac{11}{90720}\theta^4 + \frac{13}{7484400}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_3(\theta) &= \frac{2(3-\theta^2) - (6+\theta^2)\cos\theta}{6\theta^4} + i\frac{6\theta - (6+\theta^2)\sin\theta}{6\theta^4} \\ &\approx \frac{1}{24} - \frac{1}{180}\theta^2 + \frac{5}{24192}\theta^4 - \frac{1}{259200}\theta^6 + i\theta\left(\frac{7}{360} - \frac{1}{840}\theta^2 + \frac{11}{362880}\theta^4 - \frac{13}{29937600}\theta^6\right)\end{aligned}$$

The program `dftcor`, below, implements the endpoint corrections for the cubic case. Given input values of ω, Δ, a, b, and an array with the eight values $h_0, \ldots, h_3, h_{M-3}, \ldots, h_M$, it returns the real and imaginary parts of the endpoint corrections in equation (13.9.13), and the factor $W(\theta)$. The code is turgid, but only because the formulas above are complicated. The formulas have cancellations to high powers of θ. It is therefore necessary to compute the right-hand sides in double precision, even when the corrections are desired only to single precision. It is also necessary to use the series expansion for small values of θ. The optimal cross-over value of θ depends on your machine's wordlength, but you can always find it experimentally as the largest value where the two methods give identical results to machine precision.

```
      SUBROUTINE dftcor(w,delta,a,b,endpts,corre,corim,corfac)
      REAL a,b,corfac,corim,corre,delta,w,endpts(8)
        For an integral approximated by a discrete Fourier transform, this routine computes the
        correction factor that multiplies the DFT and the endpoint correction to be added. Input
        is the angular frequency w, stepsize delta, lower and upper limits of the integral a and b,
        while the array endpts contains the first 4 and last 4 function values. The correction factor
        W(θ) is returned as corfac, while the real and imaginary parts of the endpoint correction
        are returned as corre and corim.
      REAL a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c,cl,cr,s,sl,sr,t,
     *     t2,t4,t6
      DOUBLE PRECISION cth,ctth,spth2,sth,sth4i,stth,th,th2,th4,
     *     tmth2,tth4i
      th=w*delta
      if (a.ge.b.or.th.lt.0.d0.or.th.gt.3.1416d0)
     *     pause 'bad arguments to dftcor'
      if(abs(th).lt.5.d-2)then            Use series.
        t=th
        t2=t*t
        t4=t2*t2
        t6=t4*t2
        corfac=1.-(11./720.)*t4+(23./15120.)*t6
        a0r=(-2./3.)+t2/45.+(103./15120.)*t4-(169./226800.)*t6
        a1r=(7./24.)-(7./180.)*t2+(5./3456.)*t4-(7./259200.)*t6
```

```
        a2r=(-1./6.)+t2/45.-(5./6048.)*t4+t6/64800.
        a3r=(1./24.)-t2/180.+(5./24192.)*t4-t6/259200.
        a0i=t*(2./45.+(2./105.)*t2-(8./2835.)*t4+(86./467775.)*t6)
        a1i=t*(7./72.-t2/168.+(11./72576.)*t4-(13./5987520.)*t6)
        a2i=t*(-7./90.+t2/210.-(11./90720.)*t4+(13./7484400.)*t6)
        a3i=t*(7./360.-t2/840.+(11./362880.)*t4-(13./29937600.)*t6)
    else                                    Use trigonometric formulas in double precision.
        cth=cos(th)
        sth=sin(th)
        ctth=cth**2-sth**2
        stth=2.d0*sth*cth
        th2=th*th
        th4=th2*th2
        tmth2=3.d0-th2
        spth2=6.d0+th2
        sth4i=1./(6.d0*th4)
        tth4i=2.d0*sth4i
        corfac=tth4i*spth2*(3.d0-4.d0*cth+ctth)
        a0r=sth4i*(-42.d0+5.d0*th2+spth2*(8.d0*cth-ctth))
        a0i=sth4i*(th*(-12.d0+6.d0*th2)+spth2*stth)
        a1r=sth4i*(14.d0*tmth2-7.d0*spth2*cth)
        a1i=sth4i*(30.d0*th-5.d0*spth2*sth)
        a2r=tth4i*(-4.d0*tmth2+2.d0*spth2*cth)
        a2i=tth4i*(-12.d0*th+2.d0*spth2*sth)
        a3r=sth4i*(2.d0*tmth2-spth2*cth)
        a3i=sth4i*(6.d0*th-spth2*sth)
    endif
    cl=a0r*endpts(1)+a1r*endpts(2)+a2r*endpts(3)+a3r*endpts(4)
    sl=a0i*endpts(1)+a1i*endpts(2)+a2i*endpts(3)+a3i*endpts(4)
    cr=a0r*endpts(8)+a1r*endpts(7)+a2r*endpts(6)+a3r*endpts(5)
    sr=-a0i*endpts(8)-a1i*endpts(7)-a2i*endpts(6)-a3i*endpts(5)
    arg=w*(b-a)
    c=cos(arg)
    s=sin(arg)
    corre=cl+c*cr-s*sr
    corim=sl+s*cr+c*sr
    return
    END
```

Since the use of `dftcor` can be confusing, we also give an illustrative program `dftint` which uses `dftcor` to compute equation (13.9.1) for general a, b, ω, and $h(t)$. Several points within this program bear mentioning: The parameters `M` and `NDFT` correspond to M and N in the above discussion. On successive calls, we recompute the Fourier transform only if a or b has changed. (We should also recompute if $h(t)$ has changed, but `FORTRAN` doesn't provide a way for us to test this.)

Since `dftint` is designed to work for any value of ω satisfying $\omega\Delta < \pi$, not just the special values returned by the DFT (equation 13.9.12), we do polynomial interpolation of degree `MPOL` on the DFT spectrum. You should be warned that a large factor of oversampling ($N \gg M$) is required for this interpolation to be accurate. After interpolation, we add the endpoint corrections from `dftcor`, which can be evaluated for any ω.

While `dftcor` is good at what it does, `dftint` is illustrative only. It is not a general purpose program, because it does not adapt its parameters `M`, `NDFT`, `MPOL`, or its interpolation scheme, to any particular function $h(t)$. You will have to experiment with your own application.

```
    SUBROUTINE dftint(func,a,b,w,cosint,sinint)
    INTEGER M,NDFT,MPOL
    REAL a,b,cosint,sinint,w,func,TWOPI
    PARAMETER (M=64,NDFT=1024,MPOL=6,TWOPI=2.*3.14159265)
    EXTERNAL func
C   USES dftcor,func,polint,realft
```

Example program illustrating how to use the routine dftcor. The user supplies an external function func that returns the quantity $h(t)$. The routine then returns $\int_a^b \cos(\omega t)h(t)\,dt$ as cosint and $\int_a^b \sin(\omega t)h(t)\,dt$ as sinint.
Parameters: The values of M, NDFT, and MPOL are merely illustrative and should be optimized for your particular application. M is the number of subintervals, NDFT is the length of the FFT (a power of 2), and MPOL is the degree of polynomial interpolation used to obtain the desired frequency from the FFT.

```
      INTEGER init,j,nn
      REAL aold,bold,c,cdft,cerr,corfac,corim,corre,delta,en,s,
     *     sdft,serr,cpol(MPOL),data(NDFT),endpts(8),spol(MPOL),
     *     xpol(MPOL)
      SAVE init,aold,bold,delta,data,endpts
      DATA init/0/,aold/-1.e30/,bold/-1.e30/
      if (init.ne.1.or.a.ne.aold.or.b.ne.bold) then     Do we need to initialize, or is only ω
          init=1                                          changed?
          aold=a
          bold=b
          delta=(b-a)/M
          do 11 j=1,M+1                                 Load the function values into the data array.
              data(j)=func(a+(j-1)*delta)
          enddo 11
          do 12 j=M+2,NDFT                              Zero pad the rest of the data array.
              data(j)=0.
          enddo 12
          do 13 j=1,4                                   Load the endpoints.
              endpts(j)=data(j)
              endpts(j+4)=data(M-3+j)
          enddo 13
          call realft(data,NDFT,1)
            realft returns the unused value corresponding to ω_{N/2} in data(2). We actually want
            this element to contain the imaginary part corresponding to ω_0, which is zero.
          data(2)=0.
      endif
        Now interpolate on the DFT result for the desired frequency. If the frequency is an ω_n, i.e.,
        the quantity en is an integer, then cdft=data(2*en-1), sdft=data(2*en), and you could
        omit the interpolation.
      en=w*delta*NDFT/TWOPI+1.
      nn=min(max(int(en-0.5*MPOL+1.),1),NDFT/2-MPOL+1)     Leftmost point for the interpola-
      do 14 j=1,MPOL                                         tion.
          cpol(j)=data(2*nn-1)
          spol(j)=data(2*nn)
          xpol(j)=nn
          nn=nn+1
      enddo 14
      call polint(xpol,cpol,MPOL,en,cdft,cerr)
      call polint(xpol,spol,MPOL,en,sdft,serr)
      call dftcor(w,delta,a,b,endpts,corre,corim,corfac)     Now get the endpoint cor-
      cdft=cdft*corfac+corre                                   rection and the multiplica-
      sdft=sdft*corfac+corim                                   tive factor W(θ).
      c=delta*cos(w*a)                                  Finally multiply by Δ and exp(iωa).
      s=delta*sin(w*a)
      cosint=c*cdft-s*sdft
      sinint=s*cdft+c*sdft
      return
      END
```

Sometimes one is interested only in the discrete frequencies ω_m of equation (13.9.5), the ones that have integral numbers of periods in the interval $[a, b]$. For smooth $h(t)$, the value of I tends to be much smaller in magnitude at these ω's than at values in between, since the integral half-periods tend to cancel precisely. (That is why one must oversample for interpolation to be accurate: $I(\omega)$ is oscillatory with small magnitude near the ω_m's.) If you want these ω_m's without messy (and possibly inaccurate) interpolation, you have to set N to

a multiple of M (compare equations 13.9.5 and 13.9.12). In the method implemented above, however, N must be at least $M+1$, so the smallest such multiple is $2M$, resulting in a factor ~ 2 unnecessary computing. Alternatively, one can derive a formula like equation (13.9.13), but with the last sample function $h_M = h(b)$ omitted from the DFT, but included entirely in the endpoint correction for h_M. Then one can set $M = N$ (an integer power of 2) and get the special frequencies of equation (13.9.5) with no additional overhead. The modified formula is

$$\begin{aligned} I(\omega_m) = \Delta e^{i\omega_m a}\Big\{ & W(\theta)[\mathrm{DFT}(h_0 \ldots h_{M-1})]_m \\ & + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 \\ & + e^{i\omega(b-a)}\left[A(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} + \alpha_3^*(\theta)h_{M-3}\right]\Big\} \end{aligned} \tag{13.9.14}$$

where $\theta \equiv \omega_m \Delta$ and $A(\theta)$ is given by

$$A(\theta) = -\alpha_0(\theta) \tag{13.9.15}$$

for the trapezoidal case, or

$$\begin{aligned} A(\theta) &= \frac{(-6+11\theta^2)+(6+\theta^2)\cos 2\theta}{6\theta^4} - i\,\mathrm{Im}[\alpha_0(\theta)] \\ &\approx \frac{1}{3} + \frac{1}{45}\theta^2 - \frac{8}{945}\theta^4 + \frac{11}{14175}\theta^6 - i\,\mathrm{Im}[\alpha_0(\theta)] \end{aligned} \tag{13.9.16}$$

for the cubic case.

Factors like $W(\theta)$ arise naturally whenever one calculates Fourier coefficients of smooth functions, and they are sometimes called attenuation factors [1]. However, the endpoint corrections are equally important in obtaining accurate values of integrals. Narasimhan and Karthikeyan [2] have given a formula that is algebraically equivalent to our trapezoidal formula. However, their formula requires the evaluation of *two* FFTs, which is unnecessary. The basic idea used here goes back at least to Filon [3] in 1928 (before the FFT!). He used Simpson's rule (quadratic interpolation). Since this interpolation is not left-right symmetric, two Fourier transforms are required. An alternative algorithm for equation (13.9.14) has been given by Lyness in [4]; for related references, see [5]. To our knowledge, the cubic-order formulas derived here have not previously appeared in the literature.

Calculating Fourier transforms when the range of integration is $(-\infty, \infty)$ can be tricky. If the function falls off reasonably quickly at infinity, you can split the integral at a large enough value of t. For example, the integration to $+\infty$ can be written

$$\begin{aligned} \int_a^\infty e^{i\omega t}h(t)\,dt &= \int_a^b e^{i\omega t}h(t)\,dt + \int_b^\infty e^{i\omega t}h(t)\,dt \\ &= \int_a^b e^{i\omega t}h(t)\,dt - \frac{h(b)e^{i\omega b}}{i\omega} + \frac{h'(b)e^{i\omega b}}{(i\omega)^2} - \cdots \end{aligned} \tag{13.9.17}$$

The splitting point b must be chosen large enough that the remaining integral over (b, ∞) is small. Successive terms in its asymptotic expansion are found by integrating by parts. The integral over (a, b) can be done using `dftint`. You keep as many terms in the asymptotic expansion as you can easily compute. See [6] for some examples of this idea. More powerful methods, which work well for long-tailed functions but which do not use the FFT, are described in [7-9].

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), p. 88. [1]

Narasimhan, M.S. and Karthikeyan, M. 1984, *IEEE Transactions on Antennas & Propagation*, vol. 32, pp. 404–408. [2]

Filon, L.N.G. 1928, *Proceedings of the Royal Society of Edinburgh*, vol. 49, pp. 38–47. [3]

Giunta, G. and Murli, A. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 97–107. [4]

Lyness, J.N. 1987, in *Numerical Integration*, P. Keast and G. Fairweather, eds. (Dordrecht: Reidel). [5]

Pantis, G. 1975, *Journal of Computational Physics*, vol. 17, pp. 229–233. [6]

Blakemore, M., Evans, G.A., and Hyslop, J. 1976, *Journal of Computational Physics*, vol. 22, pp. 352–376. [7]

Lyness, J.N., and Kaper, T.J. 1987, *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. 1005–1011. [8]

Thakkar, A.J., and Smith, V.H. 1975, *Computer Physics Communications*, vol. 10, pp. 73–79. [9]

13.10 Wavelet Transforms

Like the fast Fourier transform (FFT), the discrete wavelet transform (DWT) is a fast, linear operation that operates on a data vector whose length is an integer power of two, transforming it into a numerically different vector of the same length. Also like the FFT, the wavelet transform is invertible and in fact orthogonal — the inverse transform, when viewed as a big matrix, is simply the transpose of the transform. Both FFT and DWT, therefore, can be viewed as a rotation in function space, from the input space (or time) domain, where the basis functions are the unit vectors $\mathbf{e}_i$, or Dirac delta functions in the continuum limit, to a different domain. For the FFT, this new domain has basis functions that are the familiar sines and cosines. In the wavelet domain, the basis functions are somewhat more complicated and have the fanciful names "mother functions" and "wavelets."

Of course there are an infinity of possible bases for function space, almost all of them uninteresting! What makes the wavelet basis interesting is that, *unlike* sines and cosines, individual wavelet functions are quite localized in space; simultaneously, *like* sines and cosines, individual wavelet functions are quite localized in frequency or (more precisely) characteristic scale. As we will see below, the particular kind of dual localization achieved by wavelets renders large classes of functions and operators sparse, or sparse to some high accuracy, when transformed into the wavelet domain. Analogously with the Fourier domain, where a class of computations, like convolutions, become computationally fast, there is a large class of computations — those that can take advantage of sparsity — that become computationally fast in the wavelet domain [1].

Unlike sines and cosines, which define a unique Fourier transform, there is not one single unique set of wavelets; in fact, there are infinitely many possible sets. Roughly, the different sets of wavelets make different trade-offs between how compactly they are localized in space and how smooth they are. (There are further fine distinctions.)

Daubechies Wavelet Filter Coefficients

A particular set of wavelets is specified by a particular set of numbers, called *wavelet filter coefficients*. Here, we will largely restrict ourselves to wavelet filters in a class discovered by Daubechies [2]. This class includes members ranging from highly localized to highly smooth. The simplest (and most localized) member, often called *DAUB4*, has only four coefficients, $c_0, \ldots, c_3$. For the moment we specialize to this case for ease of notation.

Giunta, G. and Murli, A. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 97–107. [4]

Lyness, J.N. 1987, in *Numerical Integration*, P. Keast and G. Fairweather, eds. (Dordrecht: Reidel). [5]

Pantis, G. 1975, *Journal of Computational Physics*, vol. 17, pp. 229–233. [6]

Blakemore, M., Evans, G.A., and Hyslop, J. 1976, *Journal of Computational Physics*, vol. 22, pp. 352–376. [7]

Lyness, J.N., and Kaper, T.J. 1987, *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. 1005–1011. [8]

Thakkar, A.J., and Smith, V.H. 1975, *Computer Physics Communications*, vol. 10, pp. 73–79. [9]

13.10 Wavelet Transforms

Like the fast Fourier transform (FFT), the discrete wavelet transform (DWT) is a fast, linear operation that operates on a data vector whose length is an integer power of two, transforming it into a numerically different vector of the same length. Also like the FFT, the wavelet transform is invertible and in fact orthogonal — the inverse transform, when viewed as a big matrix, is simply the transpose of the transform. Both FFT and DWT, therefore, can be viewed as a rotation in function space, from the input space (or time) domain, where the basis functions are the unit vectors $\mathbf{e}_i$, or Dirac delta functions in the continuum limit, to a different domain. For the FFT, this new domain has basis functions that are the familiar sines and cosines. In the wavelet domain, the basis functions are somewhat more complicated and have the fanciful names "mother functions" and "wavelets."

Of course there are an infinity of possible bases for function space, almost all of them uninteresting! What makes the wavelet basis interesting is that, *unlike* sines and cosines, individual wavelet functions are quite localized in space; simultaneously, *like* sines and cosines, individual wavelet functions are quite localized in frequency or (more precisely) characteristic scale. As we will see below, the particular kind of dual localization achieved by wavelets renders large classes of functions and operators sparse, or sparse to some high accuracy, when transformed into the wavelet domain. Analogously with the Fourier domain, where a class of computations, like convolutions, become computationally fast, there is a large class of computations — those that can take advantage of sparsity — that become computationally fast in the wavelet domain [1].

Unlike sines and cosines, which define a unique Fourier transform, there is not one single unique set of wavelets; in fact, there are infinitely many possible sets. Roughly, the different sets of wavelets make different trade-offs between how compactly they are localized in space and how smooth they are. (There are further fine distinctions.)

Daubechies Wavelet Filter Coefficients

A particular set of wavelets is specified by a particular set of numbers, called *wavelet filter coefficients*. Here, we will largely restrict ourselves to wavelet filters in a class discovered by Daubechies [2]. This class includes members ranging from highly localized to highly smooth. The simplest (and most localized) member, often called *DAUB4*, has only four coefficients, $c_0, \ldots, c_3$. For the moment we specialize to this case for ease of notation.

Consider the following transformation matrix acting on a column vector of data to its right:

$$\begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & & & \\
\vdots & \vdots & & & & & \ddots & & & & & \\
 & & & & & & & c_0 & c_1 & c_2 & c_3 & \\
 & & & & & & & c_3 & -c_2 & c_1 & -c_0 & \\
c_2 & c_3 & & & & & & & & c_0 & c_1 & \\
c_1 & -c_0 & & & & & & & & c_3 & -c_2 &
\end{bmatrix} \tag{13.10.1}$$

Here blank entries signify zeroes. Note the structure of this matrix. The first row generates one component of the data convolved with the filter coefficients $c_0 \ldots, c_3$. Likewise the third, fifth, and other odd rows. If the even rows followed this pattern, offset by one, then the matrix would be a circulant, that is, an ordinary convolution that could be done by FFT methods. (Note how the last two rows wrap around like convolutions with periodic boundary conditions.) Instead of convolving with $c_0, \ldots, c_3$, however, the even rows perform a different convolution, with coefficients $c_3, -c_2, c_1, -c_0$. The action of the matrix, overall, is thus to perform two related convolutions, then to decimate each of them by half (throw away half the values), and interleave the remaining halves.

It is useful to think of the filter $c_0, \ldots, c_3$ as being a smoothing filter, call it H, something like a moving average of four points. Then, because of the minus signs, the filter $c_3, -c_2, c_1, -c_0$, call it G, is *not* a smoothing filter. (In signal processing contexts, H and G are called *quadrature mirror filters* [3].) In fact, the c's are chosen so as to make G yield, insofar as possible, a *zero* response to a sufficiently smooth data vector. This is done by requiring the sequence $c_3, -c_2, c_1, -c_0$ to have a certain number of vanishing moments. When this is the case for p moments (starting with the zeroth), a set of wavelets is said to satisfy an "approximation condition of order p." This results in the output of H, decimated by half, accurately representing the data's "smooth" information. The output of G, also decimated, is referred to as the data's "detail" information [4].

For such a characterization to be useful, it must be possible to reconstruct the original data vector of length N from its $N/2$ smooth or s-components and its $N/2$ detail or d-components. That is effected by requiring the matrix (13.10.1) to be orthogonal, so that its inverse is just the transposed matrix

$$\begin{bmatrix}
c_0 & c_3 & & & \cdots & & & & & c_2 & c_1 \\
c_1 & -c_2 & & & \cdots & & & & & c_3 & -c_0 \\
c_2 & c_1 & c_0 & c_3 & & & & & & & \\
c_3 & -c_0 & c_1 & -c_2 & & & & & & & \\
 & & & & \ddots & & & & & & \\
 & & & & & c_2 & c_1 & c_0 & c_3 & & \\
 & & & & & c_3 & -c_0 & c_1 & -c_2 & & \\
 & & & & & & & c_2 & c_1 & c_0 & c_3 \\
 & & & & & & & c_3 & -c_0 & c_1 & -c_2
\end{bmatrix} \tag{13.10.2}$$

One sees immediately that matrix (13.10.2) is inverse to matrix (13.10.1) if and only if these two equations hold,

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\ c_2 c_0 + c_3 c_1 &= 0 \end{aligned} \tag{13.10.3}$$

If additionally we require the approximation condition of order $p = 2$, then two additional relations are required,

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0 \end{aligned} \tag{13.10.4}$$

Equations (13.10.3) and (13.10.4) are 4 equations for the 4 unknowns $c_0, \ldots, c_3$, first recognized and solved by Daubechies. The unique solution (up to a left-right reversal) is

$$\begin{aligned} c_0 &= (1+\sqrt{3})/4\sqrt{2} \qquad & c_1 &= (3+\sqrt{3})/4\sqrt{2} \\ c_2 &= (3-\sqrt{3})/4\sqrt{2} \qquad & c_3 &= (1-\sqrt{3})/4\sqrt{2} \end{aligned} \tag{13.10.5}$$

In fact, DAUB4 is only the most compact of a sequence of wavelet sets: If we had six coefficients instead of four, there would be three orthogonality requirements in equation (13.10.3) (with offsets of zero, two, and four), and we could require the vanishing of $p = 3$ moments in equation (13.10.4). In this case, DAUB6, the solution coefficients can also be expressed in closed form,

$$\begin{aligned} c_0 &= (1+\sqrt{10}+\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_1 &= (5+\sqrt{10}+3\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\ c_2 &= (10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_3 &= (10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\ c_4 &= (5+\sqrt{10}-3\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_5 &= (1+\sqrt{10}-\sqrt{5+2\sqrt{10}})/16\sqrt{2} \end{aligned} \tag{13.10.6}$$

For higher p, up to 10, Daubechies [2] has tabulated the coefficients numerically. The number of coefficients increases by two each time p is increased by one.

Discrete Wavelet Transform

We have not yet defined the discrete wavelet transform (DWT), but we are almost there: The DWT consists of applying a wavelet coefficient matrix like (13.10.1) *hierarchically*, first to the full data vector of length N, then to the "smooth" vector of length $N/2$, then to the "smooth-smooth" vector of length $N/4$, and so on until only a trivial number of "smooth-. . .-smooth" components (usually 2) remain. The procedure is sometimes called a *pyramidal algorithm* [4], for obvious reasons. The output of the DWT consists of these remaining components and all the "detail" components that were accumulated along the way. A diagram should make the procedure clear:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \end{bmatrix} \xrightarrow{13.10.1} \begin{bmatrix} s_1 \\ d_1 \\ s_2 \\ d_2 \\ s_3 \\ d_3 \\ s_4 \\ d_4 \\ s_5 \\ d_5 \\ s_6 \\ d_6 \\ s_7 \\ d_7 \\ s_8 \\ d_8 \end{bmatrix} \xrightarrow{\text{permute}} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ \underline{s_8} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \xrightarrow{13.10.1} \begin{bmatrix} S_1 \\ D_1 \\ S_2 \\ D_2 \\ S_3 \\ D_3 \\ S_4 \\ \underline{D_4} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \xrightarrow{\text{permute}} \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ \underline{S_4} \\ D_1 \\ D_2 \\ D_3 \\ \underline{D_4} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \xrightarrow{\text{etc.}} \begin{bmatrix} \mathcal{S}_1 \\ \underline{\mathcal{S}_2} \\ \mathcal{D}_1 \\ \underline{\mathcal{D}_2} \\ D_1 \\ D_2 \\ D_3 \\ \underline{D_4} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} \tag{13.10.7}$$

If the length of the data vector were a higher power of two, there would be more stages of applying (13.10.1) (or any other wavelet coefficients) and permuting. The endpoint will always be a vector with two $\mathcal{S}$'s and a hierarchy of $\mathcal{D}$'s, D's, d's, etc. Notice that once d's are generated, they simply propagate through to all subsequent stages.

A value d_i of any level is termed a "wavelet coefficient" of the original data vector; the final values $\mathcal{S}_1, \mathcal{S}_2$ should strictly be called "mother-function coefficients," although the term "wavelet coefficients" is often used loosely for both d's and final $\mathcal{S}$'s. Since the full procedure is a composition of orthogonal linear operations, the whole DWT is itself an orthogonal linear operator.

To invert the DWT, one simply reverses the procedure, starting with the smallest level of the hierarchy and working (in equation 13.10.7) from right to left. The inverse matrix (13.10.2) is of course used instead of the matrix (13.10.1).

As already noted, the matrices (13.10.1) and (13.10.2) embody periodic ("wrap-around") boundary conditions on the data vector. One normally accepts this as a minor inconvenience: the last few wavelet coefficients at each level of the hierarchy are affected by data from both ends of the data vector. By circularly shifting the matrix (13.10.1) $N/2$ columns to the left, one can symmetrize the wrap-around; but this does not eliminate it. It is in fact possible to eliminate the wrap-around completely by altering the coefficients in the first and last N rows of (13.10.1), giving an orthogonal matrix that is purely band-diagonal [5]. This variant, beyond our scope here, is useful when, e.g., the data varies by many orders of magnitude from one end of the data vector to the other.

Here is a routine, `wt1`, that performs the pyramidal algorithm (or its inverse if `isign` is negative) on some data vector `a(1:n)`. Successive applications of the wavelet filter, and accompanying permutations, are done by an assumed routine `wtstep`, which must be provided. (We give examples of several different `wtstep` routines just below.)

```
      SUBROUTINE wt1(a,n,isign,wtstep)
      INTEGER isign,n
      REAL a(n)
      EXTERNAL wtstep
C     USES wtstep
```

One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm, replacing `a(1:n)` by its wavelet transform (for `isign=1`), or performing the inverse operation (for `isign=-1`). Note that `n` MUST be an integer power of 2. The subroutine

wtstep, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.

```
      INTEGER nn
      if (n.lt.4) return
      if (isign.ge.0) then                  Wavelet transform.
          nn=n                              Start at largest hierarchy,
1         if (nn.ge.4) then
              call wtstep(a,nn,isign)       and work towards smallest.
              nn=nn/2
          goto 1
          endif
      else                                  Inverse wavelet transform.
          nn=4                              Start at smallest hierarchy,
2         if (nn.le.n) then
              call wtstep(a,nn,isign)
              nn=nn*2                       and work towards largest.
          goto 2
          endif
      endif
      return
      END
```

Here, as a specific instance of wtstep, is a routine for the DAUB4 wavelets:

```
      SUBROUTINE daub4(a,n,isign)
      INTEGER n,isign,NMAX          NMAX is the maximum allowed value of n.
      REAL a(n),C3,C2,C1,C0
      PARAMETER (C0=0.4829629131445341,C1=0.8365163037378079,
     *     C2=0.2241438680420134,C3=-0.1294095225512604,NMAX=1024)
        Applies the Daubechies 4-coefficient wavelet filter to data vector a(1:n) (for isign=1) or
        applies its transpose (for isign=-1). Used hierarchically by routines wt1 and wtn.
      REAL wksp(NMAX)
      INTEGER nh,nh1,i,j
      if(n.lt.4)return
      if(n.gt.NMAX) pause 'wksp too small in daub4'
      nh=n/2
      nh1=nh+1
      if (isign.ge.0) then                      Apply filter.
          i=1
          do 11 j=1,n-3,2
              wksp(i)=C0*a(j)+C1*a(j+1)+C2*a(j+2)+C3*a(j+3)
              wksp(i+nh)=C3*a(j)-C2*a(j+1)+C1*a(j+2)-C0*a(j+3)
              i=i+1
          enddo 11
          wksp(i)=C0*a(n-1)+C1*a(n)+C2*a(1)+C3*a(2)
          wksp(i+nh)=C3*a(n-1)-C2*a(n)+C1*a(1)-C0*a(2)
      else                                      Apply transpose filter.
          wksp(1)=C2*a(nh)+C1*a(n)+C0*a(1)+C3*a(nh1)
          wksp(2)=C3*a(nh)-C0*a(n)+C1*a(1)-C2*a(nh1)
          j=3
          do 12 i=1,nh-1
              wksp(j)=C2*a(i)+C1*a(i+nh)+C0*a(i+1)+C3*a(i+nh1)
              wksp(j+1)=C3*a(i)-C0*a(i+nh)+C1*a(i+1)-C2*a(i+nh1)
              j=j+2
          enddo 12
      endif
      do 13 i=1,n
          a(i)=wksp(i)
      enddo 13
      return
      END
```

For larger sets of wavelet coefficients, the wrap-around of the last rows or columns is a programming inconvenience. An efficient implementation would handle the wrap-arounds as special cases, outside of the main loop. Here, we will content ourselves with a more general scheme involving some extra arithmetic at run time. The following routine sets up any particular wavelet coefficients whose values you happen to know.

```
      SUBROUTINE pwtset(n)
      INTEGER n,NCMAX,ncof,ioff,joff
      PARAMETER (NCMAX=50)          Maximum number of wavelet coefficients passed to pwt.
      REAL cc(NCMAX),cr(NCMAX)
      COMMON /pwtcom/ cc,cr,ncof,ioff,joff
        Initializing routine for pwt, here implementing the Daubechies wavelet filters with 4, 12,
        and 20 coefficients, as selected by the input value n. Further wavelet filters can be included
        in the obvious manner. This routine must be called (once) before the first use of pwt. (For
        the case n=4, the specific routine daub4 is considerably faster than pwt.)
      INTEGER k
      REAL sig,c4(4),c12(12),c20(20)
      SAVE c4,c12,c20,/pwtcom/
      DATA c4/0.4829629131445341, 0.8365163037378079,
     *     0.2241438680420134,-0.1294095225512604/
      DATA c12 /.111540743350, .494623890398, .751133908021,
     *      .315250351709,-.226264693965,-.129766867567,
     *      .097501605587, .027522865530,-.031582039318,
     *      .000553842201, .004777257511,-.001077301085/
      DATA c20 /.026670057901, .188176800078, .527201188932,
     *      .688459039454, .281172343661,-.249846424327,
     *     -.195946274377, .127369340336, .093057364604,
     *     -.071394147166,-.029457536822, .033212674059,
     *      .003606553567,-.010733175483, .001395351747,
     *      .001992405295,-.000685856695,-.000116466855,
     *      .000093588670,-.000013264203 /
      ncof=n
      sig=-1.
      do 11 k=1,n
        if(n.eq.4)then
          cc(k)=c4(k)
        else if(n.eq.12)then
          cc(k)=c12(k)
        else if(n.eq.20)then
          cc(k)=c20(k)
        else
          pause 'unimplemented value n in pwtset'
        endif
        cr(ncof+1-k)=sig*cc(k)
        sig=-sig
      enddo 11
      ioff=-n/2                  These values center the "support" of the wavelets at each level.
      joff=-n/2                    Alternatively, the "peaks" of the wavelets can be approx-
      return                       imately centered by the choices ioff=-2 and joff=-n+2.
      END                          Note that daub4 and pwtset with n=4 use different default
                                   centerings.
```

Once pwtset has been called, the following routine can be used as a specific instance of wtstep.

```
      SUBROUTINE pwt(a,n,isign)
      INTEGER isign,n,NMAX,NCMAX,ncof,ioff,joff
      PARAMETER (NMAX=2048,NCMAX=50)
      REAL a(n),wksp(NMAX),cc(NCMAX),cr(NCMAX)
      COMMON /pwtcom/ cc,cr,ncof,ioff,joff
```

```
    Partial wavelet transform: applies an arbitrary wavelet filter to data vector a(1:n) (for
    isign=1) or applies its transpose (for isign=-1). Used hierarchically by routines wt1
    and wtn. The actual filter is determined by a preceding (and required) call to pwtset,
    which initializes the common block pwtcom.
INTEGER i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod
REAL ai,ai1
SAVE /pwtcom/
if (n.lt.4) return
nmod=ncof*n                          A positive constant equal to zero mod n.
n1=n-1                               Mask of all bits, since n a power of 2.
nh=n/2
do 11 j=1,n
    wksp(j)=0.
enddo 11
if (isign.ge.0) then                 Apply filter.
    ii=1
    do 13 i=1,n,2
        ni=i+nmod+ioff               Pointer to be incremented and wrapped-around.
        nj=i+nmod+joff
        do 12 k=1,ncof
            jf=iand(n1,ni+k)         We use bitwise and to wrap-around the pointers.
            jr=iand(n1,nj+k)
            wksp(ii)=wksp(ii)+cc(k)*a(jf+1)
            wksp(ii+nh)=wksp(ii+nh)+cr(k)*a(jr+1)
        enddo 12
        ii=ii+1
    enddo 13
else                                 Apply transpose filter.
    ii=1
    do 15 i=1,n,2
        ai=a(ii)
        ai1=a(ii+nh)
        ni=i+nmod+ioff               See comments above.
        nj=i+nmod+joff
        do 14 k=1,ncof
            jf=iand(n1,ni+k)+1
            jr=iand(n1,nj+k)+1
            wksp(jf)=wksp(jf)+cc(k)*ai
            wksp(jr)=wksp(jr)+cr(k)*ai1
        enddo 14
        ii=ii+1
    enddo 15
endif
do 16 j=1,n                          Copy the results back from workspace.
    a(j)=wksp(j)
enddo 16
return
END
```

What Do Wavelets Look Like?

We are now in a position actually to see some wavelets. To do so, we simply run unit vectors through any of the above discrete wavelet transforms, with `isign` negative so that the inverse transform is performed. Figure 13.10.1 shows the DAUB4 wavelet that is the inverse DWT of a unit vector in the 5th component of a vector of length 1024, and also the DAUB20 wavelet that is the inverse of the 22nd component. (One needs to go to a later hierarchical level for DAUB20, to avoid a wavelet with a wrapped-around tail.) Other unit vectors would give wavelets with the same shapes, but different positions and scales.

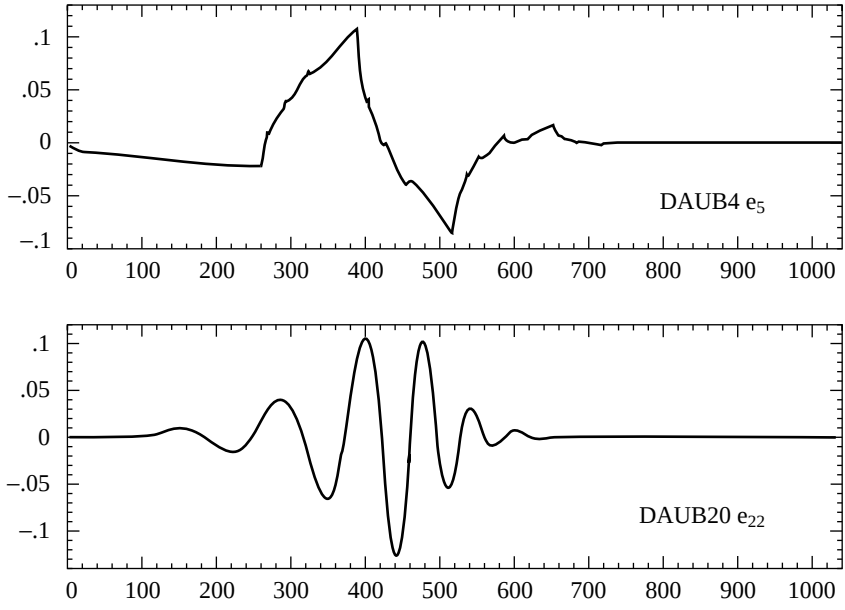


Figure 13.10.1. Wavelet functions, that is, single basis functions from the wavelet families DAUB4 and DAUB20. A complete, orthonormal wavelet basis consists of scalings and translations of either one of these functions. DAUB4 has an infinite number of cusps; DAUB20 would show similar behavior in a higher derivative.

One sees that both DAUB4 and DAUB20 have wavelets that are continuous. DAUB20 wavelets also have higher continuous derivatives. DAUB4 has the peculiar property that its derivative exists only *almost* everywhere. Examples of where it fails to exist are the points $p/2^n$, where p and n are integers; at such points, DAUB4 is left differentiable, but not right differentiable! This kind of discontinuity — at least in some derivative — is a necessary feature of wavelets with compact support, like the Daubechies series. For every increase in the number of wavelet coefficients by two, the Daubechies wavelets gain about *half* a derivative of continuity. (But not exactly half; the actual orders of regularity are irrational numbers!)

Note that the fact that wavelets are not smooth does not prevent their having exact representations for some smooth functions, as demanded by their approximation order p. The continuity of a wavelet is not the same as the continuity of functions that a set of wavelets can represent. For example, DAUB4 can represent (piecewise) linear functions of arbitrary slope: in the correct linear combinations, the cusps all cancel out. Every increase of two in the number of coefficients allows one higher order of polynomial to be exactly represented.

Figure 13.10.2 shows the result of performing the inverse DWT on the input vector $\mathbf{e}_{10} + \mathbf{e}_{58}$, again for the two different particular wavelets. Since 10 lies early in the hierarchical range of $9 - 16$, that wavelet lies on the left side of the picture. Since 58 lies in a later (smaller-scale) hierarchy, it is a narrower wavelet; in the range of 33–64 it is towards the end, so it lies on the right side of the picture. Note that smaller-scale wavelets are taller, so as to have the same squared integral.

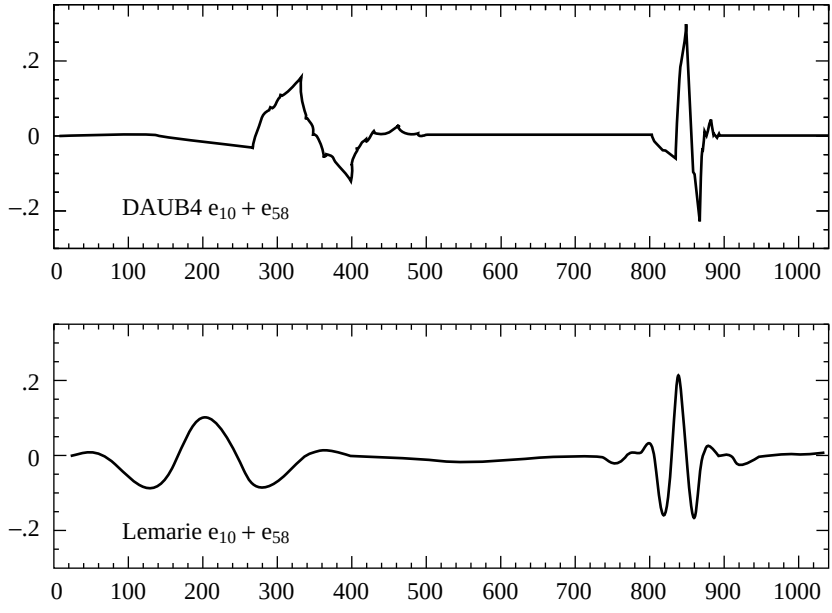


Figure 13.10.2. More wavelets, here generated from the sum of two unit vectors, $\mathbf{e}_{10} + \mathbf{e}_{58}$, which are in different hierarchical levels of scale, and also at different spatial positions. DAUB4 wavelets (a) are defined by a filter in coordinate space (equation 13.10.5), while Lemarie wavelets (b) are defined by a filter most easily written in Fourier space (equation 13.10.14).

Wavelet Filters in the Fourier Domain

The Fourier transform of a set of filter coefficients c_j is given by

$$H(\omega) = \sum_j c_j e^{ij\omega} \tag{13.10.8}$$

Here H is a function periodic in 2π, and it has the same meaning as before: It is the wavelet filter, now written in the Fourier domain. A very useful fact is that the orthogonality conditions for the c's (e.g., equation 13.10.3 above) collapse to two simple relations in the Fourier domain,

$$\frac{1}{2}|H(0)|^2 = 1 \tag{13.10.9}$$

and

$$\frac{1}{2}\left[|H(\omega)|^2 + |H(\omega+\pi)|^2\right] = 1 \tag{13.10.10}$$

Likewise the approximation condition of order p (e.g., equation 13.10.4 above) has a simple formulation, requiring that $H(\omega)$ have a pth order zero at $\omega = \pi$, or (equivalently)

$$H^{(m)}(\pi) = 0 \qquad m = 0, 1, \ldots, p-1 \tag{13.10.11}$$

It is thus relatively straightforward to invent wavelet sets in the Fourier domain. You simply invent a function $H(\omega)$ satisfying equations (13.10.9)–(13.10.11). To find the actual c_j's applicable to a data (or s-component) vector of length N, and with periodic wrap-around as in matrices (13.10.1) and (13.10.2), you invert equation (13.10.8) by the discrete Fourier transform

$$c_j = \frac{1}{N}\sum_{k=0}^{N-1} H(\frac{2\pi k}{N})e^{-2\pi ijk/N} \tag{13.10.12}$$

The quadrature mirror filter G (reversed c_j's with alternating signs), incidentally, has the Fourier representation

$$G(\omega) = e^{-i\omega}H^*(\omega+\pi) \tag{13.10.13}$$

where asterisk denotes complex conjugation.

In general the above procedure will *not* produce wavelet filters with compact support. In other words, all N of the c_j's, $j = 0, \ldots, N-1$ will in general be nonzero (though they may be rapidly decreasing in magnitude). The Daubechies wavelets, or other wavelets with compact support, are specially chosen so that $H(\omega)$ is a trigonometric polynomial with only a small number of Fourier components, guaranteeing that there will be only a small number of nonzero c_j's.

On the other hand, there is sometimes no particular reason to demand compact support. Giving it up in fact allows the ready construction of relatively smoother wavelets (higher values of p). Even without compact support, the convolutions implicit in the matrix (13.10.1) can be done efficiently by FFT methods.

Lemarie's wavelet (see [4]) has $p = 4$, does not have compact support, and is defined by the choice of $H(\omega)$,

$$H(\omega) = \left[2(1-u)^4\frac{315-420u+126u^2-4u^3}{315-420v+126v^2-4v^3}\right]^{1/2} \tag{13.10.14}$$

where

$$u \equiv \sin^2\frac{\omega}{2} \qquad v \equiv \sin^2\omega \tag{13.10.15}$$

It is beyond our scope to explain where equation (13.10.14) comes from. An informal description is that the quadrature mirror filter $G(\omega)$ deriving from equation (13.10.14) has the property that it gives identically zero when applied to any function whose odd-numbered samples are equal to the cubic spline interpolation of its even-numbered samples. Since this class of functions includes many very smooth members, it follows that $H(\omega)$ does a good job of truly selecting a function's smooth information content. Sample Lemarie wavelets are shown in Figure 13.10.2.

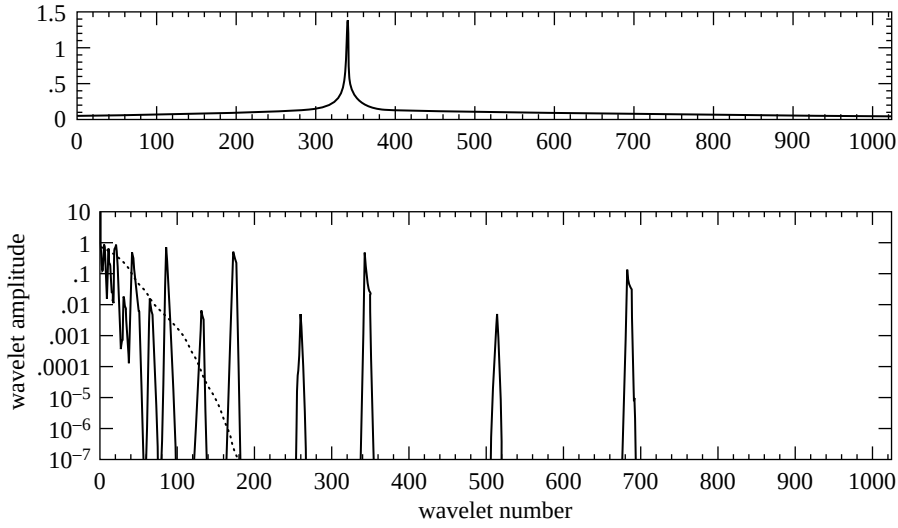


Figure 13.10.3. (a) Arbitrary test function, with cusp, sampled on a vector of length 1024. (b) Absolute value of the 1024 wavelet coefficients produced by the discrete wavelet transform of (a). Note log scale. The dotted curve plots the same amplitudes when sorted by decreasing size. One sees that only 130 out of 1024 coefficients are larger than 10^{-4} (or larger than about 10^{-5} times the largest coefficient, whose value is ~ 10).

Truncated Wavelet Approximations

Most of the usefulness of wavelets rests on the fact that wavelet transforms can usefully be severely truncated, that is, turned into sparse expansions. The case of Fourier transforms is different: FFTs are ordinarily used without truncation, to compute fast convolutions, for example. This works because the convolution operator is particularly simple in the Fourier basis. There are not, however, any standard mathematical operations that are especially simple in the wavelet basis.

To see how truncation works, consider the simple example shown in Figure 13.10.3. The upper panel shows an arbitrarily chosen test function, smooth except for a square-root cusp, sampled onto a vector of length 2^{10}. The bottom panel (solid curve) shows, on a log scale, the absolute value of the vector's components after it has been run through the DAUB4 discrete wavelet transform. One notes, from right to left, the different levels of hierarchy, 513–1024, 257–512, 129–256, etc. Within each level, the wavelet coefficients are non-negligible only very near the location of the cusp, or very near the left and right boundaries of the hierarchical range (edge effects).

The dotted curve in the lower panel of Figure 13.10.3 plots the same amplitudes as the solid curve, but sorted into decreasing order of size. One can read off, for example, that the 130th largest wavelet coefficient has an amplitude less than 10^{-5} of the largest coefficient, whose magnitude is ~ 10 (power or square integral ratio less than 10^{-10}). Thus, the example function can be represented quite accurately by only 130, rather than 1024, coefficients — the remaining ones being set to zero. Note that this kind of truncation makes the vector sparse, but not shorter than 1024. It is very important that vectors in wavelet space be truncated according to the *amplitude* of the components, not their position in the vector. Keeping the first 256

components of the vector (all levels of the hierarchy except the last two) would give an extremely poor, and jagged, approximation to the function. When you compress a function with wavelets, you have to record both the values *and the positions* of the nonzero coefficients.

Generally, compact (and therefore unsmooth) wavelets are better for lower accuracy approximation and for functions with discontinuities (like edges), while smooth (and therefore noncompact) wavelets are better for achieving high numerical accuracy. This makes compact wavelets a good choice for image compression, for example, while it makes smooth wavelets best for fast solution of integral equations.

Wavelet Transform in Multidimensions

A wavelet transform of a d-dimensional array is most easily obtained by transforming the array sequentially on its first index (for all values of its other indices), then on its second, and so on. Each transformation corresponds to multiplication by an orthogonal matrix. By matrix associativity, the result is independent of the order in which the indices were transformed. The situation is exactly like that for multidimensional FFTs. A routine for effecting the multidimensional DWT can thus be modeled on a multidimensional FFT routine like `fourn`:

```
      SUBROUTINE wtn(a,nn,ndim,isign,wtstep)
      INTEGER isign,ndim,nn(ndim),NMAX
      REAL a(*)
      EXTERNAL wtstep
      PARAMETER (NMAX=1024)
C     USES wtstep
```

Replaces `a` by its `ndim`-dimensional discrete wavelet transform, if `isign` is input as 1. `nn` is an integer array of length `ndim`, containing the lengths of each dimension (number of real values), which MUST all be powers of 2. `a` is a real array of length equal to the product of these lengths, in which the data are stored as in a multidimensional real `FORTRAN` array. If `isign` is input as -1, `a` is replaced by its inverse wavelet transform. The subroutine `wtstep`, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of `wtstep` are `daub4` and (preceded by `pwtset`) `pwt`.

```
      INTEGER i1,i2,i3,idim,k,n,nnew,nprev,nt,ntot
      REAL wksp(NMAX)
      ntot=1
      do 11 idim=1,ndim
        ntot=ntot*nn(idim)
      enddo 11
      nprev=1
      do 16 idim=1,ndim                          Main loop over the dimensions.
        n=nn(idim)
        nnew=n*nprev
        if (n.gt.4) then
          do 15 i2=0,ntot-1,nnew
            do 14 i1=1,nprev
              i3=i1+i2
              do 12 k=1,n                        Copy the relevant row or column or etc. into
                wksp(k)=a(i3)                      workspace.
                i3=i3+nprev
              enddo 12
              if (isign.ge.0) then               Do one-dimensional wavelet transform.
                nt=n
1               if (nt.ge.4) then
                call wtstep(wksp,nt,isign)
                nt=nt/2
                goto 1
                endif
```

```
                else                              Or inverse transform.
                    nt=4
2                   if (nt.le.n) then
                    call wtstep(wksp,nt,isign)
                    nt=nt*2
                    goto 2
                    endif
                endif
                i3=i1+i2
                do 13 k=1,n                       Copy back from workspace.
                    a(i3)=wksp(k)
                    i3=i3+nprev
                enddo 13
            enddo 14
        enddo 15
    endif
    nprev=nnew
enddo 16
return
END
```

Here, as before, `wtstep` is an individual wavelet step, either `daub4` or `pwt`.

Compression of Images

An immediate application of the multidimensional transform `wtn` is to image compression. The overall procedure is to take the wavelet transform of a digitized image, and then to "allocate bits" among the wavelet coefficients in some highly nonuniform, optimized, manner. In general, large wavelet coefficients get quantized accurately, while small coefficients are quantized coarsely with only a bit or two — or else are truncated completely. If the resulting quantization levels are still statistically nonuniform, they may then be further compressed by a technique like Huffman coding (§20.4).

While a more detailed description of the "back end" of this process, namely the quantization and coding of the image, is beyond our scope, it is quite straightforward to demonstrate the "front-end" wavelet encoding with a simple truncation: We keep (with full accuracy) all wavelet coefficients larger than some threshold, and we delete (set to zero) all smaller wavelet coefficients. We can then adjust the threshold to vary the fraction of preserved coefficients.

Figure 13.10.4 shows a sequence of images that differ in the number of wavelet coefficients that have been kept. The original picture (a), which is an official IEEE test image, has 256 by 256 pixels with an 8-bit grayscale. The two reproductions following are reconstructed with 23% (b) and 5.5% (c) of the 65536 wavelet coefficients. The latter image illustrates the kind of compromises made by the truncated wavelet representation. High-contrast edges (the model's right cheek and hair highlights, e.g.) are maintained at a relatively high resolution, while low-contrast areas (the model's left eye and cheek, e.g.) are washed out into what amounts to large constant pixels. Figure 13.10.4 (d) is the result of performing the identical procedure with Fourier, instead of wavelet, transforms: The figure is reconstructed from the 5.5% of 65536 real Fourier components having the largest magnitudes. One sees that, since sines and cosines are nonlocal, the resolution is uniformly poor across the picture; also, the deletion of any components produces a mottled "ringing" everywhere. (Practical Fourier image compression schemes therefore break up an

Figure 13.10.4. (a) IEEE test image, 256×256 pixels with 8-bit grayscale. (b) The image is transformed into the wavelet basis; 77% of its wavelet components are set to zero (those of smallest magnitude); it is then reconstructed from the remaining 23%. (c) Same as (b), but 94.5% of the wavelet components are deleted. (d) Same as (c), but the Fourier transform is used instead of the wavelet transform. Wavelet coefficients are better than the Fourier coefficients at preserving relevant details.

image into small blocks of pixels, 16×16, say, and do rather elaborate smoothing across block boundaries when the image is reconstructed.)

Fast Solution of Linear Systems

One of the most interesting, and promising, wavelet applications is linear algebra. The basic idea [1] is to think of an integral operator (that is, a large matrix) as a digital image. Suppose that the operator compresses well under a two-dimensional wavelet transform, i.e., that a large fraction of its wavelet coefficients are so small as to be negligible. Then any linear system involving the operator becomes a sparse

system in the wavelet basis. In other words, to solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{13.10.16}$$

we first wavelet-transform the operator $\mathbf{A}$ and the right-hand side $\mathbf{b}$ by

$$\widetilde{\mathbf{A}} \equiv \mathbf{W} \cdot \mathbf{A} \cdot \mathbf{W}^T, \qquad \widetilde{\mathbf{b}} \equiv \mathbf{W} \cdot \mathbf{b} \tag{13.10.17}$$

where $\mathbf{W}$ represents the one-dimensional wavelet transform, then solve

$$\widetilde{\mathbf{A}} \cdot \widetilde{\mathbf{x}} = \widetilde{\mathbf{b}} \tag{13.10.18}$$

and finally transform to the answer by the inverse wavelet transform

$$\mathbf{x} = \mathbf{W}^T \cdot \widetilde{\mathbf{x}} \tag{13.10.19}$$

(Note that the routine `wtn` does the complete transformation of $\mathbf{A}$ into $\widetilde{\mathbf{A}}$.)

A typical integral operator that compresses well into wavelets has arbitrary (or even nearly singular) elements near to its main diagonal, but becomes smooth away from the diagonal. An example might be

$$A_{ij} = \begin{cases} -1 & \text{if } i = j \\ |i-j|^{-1/2} & \text{otherwise} \end{cases} \tag{13.10.20}$$

Figure 13.10.5 shows a graphical representation of the wavelet transform of this matrix, where i and j range over $1 \ldots 256$, using the DAUB12 wavelets. Elements larger in magnitude than 10^{-3} times the maximum element are shown as black pixels, while elements between 10^{-3} and 10^{-6} are shown in gray. White pixels are $< 10^{-6}$. The indices i and j each number from the lower left.

In the figure, one sees the hierarchical decomposition into power-of-two sized blocks. At the edges or corners of the various blocks, one sees edge effects caused by the wrap-around wavelet boundary conditions. Apart from edge effects, within each block, the nonnegligible elements are concentrated along the block diagonals. This is a statement that, for this type of linear operator, a wavelet is coupled mainly to near neighbors in its own hierarchy (square blocks along the main diagonal) and near neighbors in other hierarchies (rectangular blocks off the diagonal).

The number of nonnegligible elements in a matrix like that in Figure 13.10.5 scales only as N, the linear size of the matrix; as a rough rule of thumb it is about $10N \log_{10}(1/\epsilon)$, where ϵ is the truncation level, e.g., 10^{-6}. For a 2000 by 2000 matrix, then, the matrix is sparse by a factor on the order of 30.

Various numerical schemes can be used to solve sparse linear systems of this "hierarchically band diagonal" form. Beylkin, Coifman, and Rokhlin [1] make the interesting observations that (1) the product of two such matrices is itself hierarchically band diagonal (truncating, of course, newly generated elements that are smaller than the predetermined threshold ϵ); and moreover that (2) the product can be formed in order N operations.

Fast matrix multiplication makes it possible to find the matrix inverse by Schultz's (or Hotelling's) method, see §2.5.

Figure 13.10.5. Wavelet transform of a 256×256 matrix, represented graphically. The original matrix has a discontinuous cusp along its diagonal, decaying smoothly away on both sides of the diagonal. In wavelet basis, the matrix becomes sparse: Components larger than 10^{-3} are shown as black, components larger than 10^{-6} as gray, and smaller-magnitude components are white. The matrix indices i and j number from the lower left.

Other schemes are also possible for fast solution of hierarchically band diagonal forms. For example, one can use the conjugate gradient method, implemented in §2.7 as `linbcg`.

CITED REFERENCES AND FURTHER READING:

Daubechies, I. 1992, *Wavelets* (Philadelphia: S.I.A.M.).

Strang, G. 1989, *SIAM Review*, vol. 31, pp. 614–627.

Beylkin, G., Coifman, R., and Rokhlin, V. 1991, *Communications on Pure and Applied Mathematics*, vol. 44, pp. 141–183. [1]

Daubechies, I. 1988, *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909–996. [2]

Vaidyanathan, P.P. 1990, *Proceedings of the IEEE*, vol. 78, pp. 56–93. [3]

Mallat, S.G. 1989, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693. [4]

Freedman, M.H., and Press, W.H. 1992, preprint. [5]

13.11 Numerical Use of the Sampling Theorem

In §6.10 we implemented an approximating formula for Dawson's integral due to Rybicki. Now that we have become Fourier sophisticates, we can learn that the formula derives from *numerical* application of the sampling theorem (§12.1), normally considered to be a purely analytic tool. Our discussion is identical to Rybicki [1].

For present purposes, the sampling theorem is most conveniently stated as follows: Consider an arbitrary function $g(t)$ and the grid of sampling points $t_n = \alpha + nh$, where n ranges over the integers and α is a constant that allows an arbitrary shift of the sampling grid. We then write

$$g(t) = \sum_{n=-\infty}^{\infty} g(t_n) \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.1}$$

where $\operatorname{sinc} x \equiv \sin x / x$. The summation over the sampling points is called the *sampling representation* of $g(t)$, and $e(t)$ is its error term. The sampling theorem asserts that the sampling representation is exact, that is, $e(t) \equiv 0$, if the Fourier transform of $g(t)$,

$$G(\omega) = \int_{-\infty}^{\infty} g(t) e^{i\omega t}\, dt \tag{13.11.2}$$

vanishes identically for $|\omega| \geq \pi/h$.

When can sampling representations be used to advantage for the approximate numerical computation of functions? In order that the error term be small, the Fourier transform $G(\omega)$ must be sufficiently small for $|\omega| \geq \pi/h$. On the other hand, in order for the summation in (13.11.1) to be approximated by a reasonably small number of terms, the function $g(t)$ itself should be very small outside of a fairly limited range of values of t. Thus we are led to two conditions to be satisfied in order that (13.11.1) be useful numerically: Both the function $g(t)$ and its Fourier transform $G(\omega)$ must rapidly approach zero for large values of their respective arguments.

Unfortunately, these two conditions are mutually antagonistic — the Uncertainty Principle in quantum mechanics. There exist strict limits on how rapidly the simultaneous approach to zero can be in both arguments. According to a theorem of Hardy [2], if $g(t) = O(e^{-t^2})$ as $|t| \to \infty$ and $G(\omega) = O(e^{-\omega^2/4})$ as $|\omega| \to \infty$, then $g(t) \equiv Ce^{-t^2}$, where C is a constant. This can be interpreted as saying that of all functions the Gaussian is the most rapidly decaying in both t and ω, and in this sense is the "best" function to be expressed numerically as a sampling representation.

Let us then write for the Gaussian $g(t) = e^{-t^2}$,

$$e^{-t^2} = \sum_{n=-\infty}^{\infty} e^{-t_n^2} \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.3}$$

The error $e(t)$ depends on the parameters h and α as well as on t, but it is sufficient for the present purposes to state the bound,

$$|e(t)| < e^{-(\pi/2h)^2} \tag{13.11.4}$$

which can be understood simply as the order of magnitude of the Fourier transform of the Gaussian at the point where it "spills over" into the region $|\omega| > \pi/h$.

When the summation in (13.11.3) is approximated by one with finite limits, say from $N_0 - N$ to $N_0 + N$, where N_0 is the integer nearest to $-\alpha/h$, there is a further truncation error. However, if N is chosen so that $N > \pi/(2h^2)$, the truncation error in the summation is less than the bound given by (13.11.4), and, since this bound is an overestimate, we shall continue to use it for (13.11.3) as well. The truncated summation gives a remarkably accurate representation for the Gaussian even for moderate values of N. For example, $|e(t)| < 5 \times 10^{-5}$ for $h = 1/2$ and $N = 7$; $|e(t)| < 2 \times 10^{-10}$ for $h = 1/3$ and $N = 15$; and $|e(t)| < 7 \times 10^{-18}$ for $h = 1/4$ and $N = 25$.

13.11 Numerical Use of the Sampling Theorem

In §6.10 we implemented an approximating formula for Dawson's integral due to Rybicki. Now that we have become Fourier sophisticates, we can learn that the formula derives from *numerical* application of the sampling theorem (§12.1), normally considered to be a purely analytic tool. Our discussion is identical to Rybicki [1].

For present purposes, the sampling theorem is most conveniently stated as follows: Consider an arbitrary function $g(t)$ and the grid of sampling points $t_n = \alpha + nh$, where n ranges over the integers and α is a constant that allows an arbitrary shift of the sampling grid. We then write

$$g(t) = \sum_{n=-\infty}^{\infty} g(t_n) \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.1}$$

where $\operatorname{sinc} x \equiv \sin x / x$. The summation over the sampling points is called the *sampling representation* of $g(t)$, and $e(t)$ is its error term. The sampling theorem asserts that the sampling representation is exact, that is, $e(t) \equiv 0$, if the Fourier transform of $g(t)$,

$$G(\omega) = \int_{-\infty}^{\infty} g(t) e^{i\omega t}\, dt \tag{13.11.2}$$

vanishes identically for $|\omega| \geq \pi/h$.

When can sampling representations be used to advantage for the approximate numerical computation of functions? In order that the error term be small, the Fourier transform $G(\omega)$ must be sufficiently small for $|\omega| \geq \pi/h$. On the other hand, in order for the summation in (13.11.1) to be approximated by a reasonably small number of terms, the function $g(t)$ itself should be very small outside of a fairly limited range of values of t. Thus we are led to two conditions to be satisfied in order that (13.11.1) be useful numerically: Both the function $g(t)$ and its Fourier transform $G(\omega)$ must rapidly approach zero for large values of their respective arguments.

Unfortunately, these two conditions are mutually antagonistic — the Uncertainty Principle in quantum mechanics. There exist strict limits on how rapidly the simultaneous approach to zero can be in both arguments. According to a theorem of Hardy [2], if $g(t) = O(e^{-t^2})$ as $|t| \to \infty$ and $G(\omega) = O(e^{-\omega^2/4})$ as $|\omega| \to \infty$, then $g(t) \equiv Ce^{-t^2}$, where C is a constant. This can be interpreted as saying that of all functions the Gaussian is the most rapidly decaying in both t and ω, and in this sense is the "best" function to be expressed numerically as a sampling representation.

Let us then write for the Gaussian $g(t) = e^{-t^2}$,

$$e^{-t^2} = \sum_{n=-\infty}^{\infty} e^{-t_n^2} \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.3}$$

The error $e(t)$ depends on the parameters h and α as well as on t, but it is sufficient for the present purposes to state the bound,

$$|e(t)| < e^{-(\pi/2h)^2} \tag{13.11.4}$$

which can be understood simply as the order of magnitude of the Fourier transform of the Gaussian at the point where it "spills over" into the region $|\omega| > \pi/h$.

When the summation in (13.11.3) is approximated by one with finite limits, say from $N_0 - N$ to $N_0 + N$, where N_0 is the integer nearest to $-\alpha/h$, there is a further truncation error. However, if N is chosen so that $N > \pi/(2h^2)$, the truncation error in the summation is less than the bound given by (13.11.4), and, since this bound is an overestimate, we shall continue to use it for (13.11.3) as well. The truncated summation gives a remarkably accurate representation for the Gaussian even for moderate values of N. For example, $|e(t)| < 5 \times 10^{-5}$ for $h = 1/2$ and $N = 7$; $|e(t)| < 2 \times 10^{-10}$ for $h = 1/3$ and $N = 15$; and $|e(t)| < 7 \times 10^{-18}$ for $h = 1/4$ and $N = 25$.

One may ask, what is the point of such a numerical representation for the Gaussian, which can be computed so easily and quickly as an exponential? The answer is that many transcendental functions can be expressed as an integral involving the Gaussian, and by substituting (13.11.3) one can often find excellent approximations to the integrals as a sum over elementary functions.

Let us consider as an example the function $w(z)$ of the complex variable $z = x + iy$, related to the complex error function by

$$w(z) = e^{-z^2} \operatorname{erfc}(-iz) \tag{13.11.5}$$

having the integral representation

$$w(z) = \frac{1}{\pi i} \int_C \frac{e^{-t^2}\, dt}{t - z} \tag{13.11.6}$$

where the contour C extends from $-\infty$ to ∞, passing below z (see, e.g., [3]). Many methods exist for the evaluation of this function (e.g., [4]). Substituting the sampling representation (13.11.3) into (13.11.6) and performing the resulting elementary contour integrals, we obtain

$$w(z) \approx \frac{1}{\pi i} \sum_{n=-\infty}^{\infty} h e^{-t_n^2} \frac{1 - (-1)^n e^{-\pi i(\alpha - z)/h}}{t_n - z} \tag{13.11.7}$$

where we now omit the error term. One should note that there is no singularity as $z \to t_m$ for some $n = m$, but a special treatment of the mth term will be required in this case (for example, by power series expansion).

An alternative form of equation (13.11.7) can be found by expressing the complex exponential in (13.11.7) in terms of trigonometric functions and using the sampling representation (13.11.3) with z replacing t. This yields

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i} \sum_{n=-\infty}^{\infty} h e^{-t_n^2} \frac{1 - (-1)^n \cos \pi(\alpha - z)/h}{t_n - z} \tag{13.11.8}$$

This form is particularly useful in obtaining Re $w(z)$ when $|y| \ll 1$. Note that in evaluating (13.11.7) the exponential inside the summation is a constant and needs to be evaluated only once; a similar comment holds for the cosine in (13.11.8).

There are a variety of formulas that can now be derived from either equation (13.11.7) or (13.11.8) by choosing particular values of α. Eight interesting choices are: $\alpha = 0$, x, iy, or z, plus the values obtained by adding $h/2$ to each of these. Since the error bound (13.11.3) assumed a real value of α, the choices involving a complex α are useful only if the imaginary part of z is not too large. This is not the place to catalog all sixteen possible formulas, and we give only two particular cases that show some of the important features.

First of all let $\alpha = 0$ in equation (13.11.8), which yields,

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i} \sum_{n=-\infty}^{\infty} h e^{-(nh)^2} \frac{1 - (-1)^n \cos(\pi z/h)}{nh - z} \tag{13.11.9}$$

This approximation is good over the entire z-plane. As stated previously, one has to treat the case where one denominator becomes small by expansion in a power series. Formulas for the case $\alpha = 0$ were discussed briefly in [5]. They are similar, but not identical, to formulas derived by Chiarella and Reichel [6], using the method of Goodwin [7].

Next, let $\alpha = z$ in (13.11.7), which yields

$$w(z) \approx e^{-z^2} - \frac{2}{\pi i} \sum_{n \text{ odd}} \frac{e^{-(z - nh)^2}}{n} \tag{13.11.10}$$

the sum being over all odd integers (positive and negative). Note that we have made the substitution $n \to -n$ in the summation. This formula is simpler than (13.11.9) and contains half the number of terms, but its error is worse if y is large. Equation (13.11.10) is the source of the approximation formula (6.10.3) for Dawson's integral, used in §6.10.

CITED REFERENCES AND FURTHER READING:

Rybicki, G.B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85–87. [1]

Hardy, G.H. 1933, *Journal of the London Mathematical Society*, vol. 8, pp. 227–231. [2]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [3]

Gautschi, W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 187–198. [4]

Armstrong, B.H., and Nicholls, R.W. 1972, *Emission, Absorption and Transfer of Radiation in Heated Atmospheres* (New York: Pergamon). [5]

Chiarella, C., and Reichel, A. 1968, *Mathematics of Computation*, vol. 22, pp. 137–143. [6]

Goodwin, E.T. 1949, *Proceedings of the Cambridge Philosophical Society*, vol. 45, pp. 241–245. [7]

Chapter 14. Statistical Description of Data

14.0 Introduction

In this chapter and the next, the concept of *data* enters the discussion more prominently than before.

Data consist of numbers, of course. But these numbers are fed into the computer, not produced by it. These are numbers to be treated with considerable respect, neither to be tampered with, nor subjected to a numerical process whose character you do not completely understand. You are well advised to acquire a reverence for data that is rather different from the "sporty" attitude that is sometimes allowable, or even commendable, in other numerical tasks.

The analysis of data inevitably involves some trafficking with the field of *statistics*, that gray area which is not quite a branch of mathematics — and just as surely not quite a branch of science. In the following sections, you will repeatedly encounter the following paradigm:

- apply some formula to the data to compute "a statistic"
- compute where the value of that statistic falls in a probability distribution that is computed on the basis of some "null hypothesis"
- if it falls in a very unlikely spot, way out on a tail of the distribution, conclude that the null hypothesis is *false* for your data set

If a statistic falls in a *reasonable* part of the distribution, you must not make the mistake of concluding that the null hypothesis is "verified" or "proved." That is the curse of statistics, that it can never prove things, only disprove them! At best, you can substantiate a hypothesis by ruling out, statistically, a whole long list of competing hypotheses, every one that has ever been proposed. After a while your adversaries and competitors will give up trying to think of alternative hypotheses, or else they will grow old and die, and *then your hypothesis will become accepted.* Sounds crazy, we know, but that's how science works!

In this book we make a somewhat arbitrary distinction between data analysis procedures that are *model-independent* and those that are *model-dependent*. In the former category, we include so-called *descriptive statistics* that characterize a data set in general terms: its mean, variance, and so on. We also include statistical tests that seek to establish the "sameness" or "differentness" of two or more data sets, or that seek to establish and measure a degree of *correlation* between two data sets. These subjects are discussed in this chapter.

In the other category, model-dependent statistics, we lump the whole subject of fitting data to a theory, parameter estimation, least-squares fits, and so on. Those subjects are introduced in Chapter 15.

Section 14.1 deals with so-called *measures of central tendency*, the moments of a distribution, the median and mode. In §14.2 we learn to test whether different data sets are drawn from distributions with different values of these measures of central tendency. This leads naturally, in §14.3, to the more general question of whether two distributions can be shown to be (significantly) different.

In §14.4–§14.7, we deal with *measures of association* for two distributions. We want to determine whether two variables are "correlated" or "dependent" on one another. If they are, we want to characterize the degree of correlation in some simple ways. The distinction between parametric and nonparametric (rank) methods is emphasized.

Section 14.8 introduces the concept of data smoothing, and discusses the particular case of Savitzky-Golay smoothing filters.

This chapter draws mathematically on the material on special functions that was presented in Chapter 6, especially §6.1–§6.4. You may wish, at this point, to review those sections.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill).

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*].

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

14.1 Moments of a Distribution: Mean, Variance, Skewness, and So Forth

When a set of values has a sufficiently strong central tendency, that is, a tendency to cluster around some particular value, then it may be useful to characterize the set by a few numbers that are related to its *moments*, the sums of integer powers of the values.

Best known is the *mean* of the values $x_1, \ldots, x_N$,

$$\overline{x} = \frac{1}{N} \sum_{j=1}^{N} x_j \tag{14.1.1}$$

which estimates the value around which central clustering occurs. Note the use of an overbar to denote the mean; angle brackets are an equally common notation, e.g., $\langle x \rangle$. You should be aware that the mean is not the only available estimator of this

In the other category, model-dependent statistics, we lump the whole subject of fitting data to a theory, parameter estimation, least-squares fits, and so on. Those subjects are introduced in Chapter 15.

Section 14.1 deals with so-called *measures of central tendency*, the moments of a distribution, the median and mode. In §14.2 we learn to test whether different data sets are drawn from distributions with different values of these measures of central tendency. This leads naturally, in §14.3, to the more general question of whether two distributions can be shown to be (significantly) different.

In §14.4–§14.7, we deal with *measures of association* for two distributions. We want to determine whether two variables are "correlated" or "dependent" on one another. If they are, we want to characterize the degree of correlation in some simple ways. The distinction between parametric and nonparametric (rank) methods is emphasized.

Section 14.8 introduces the concept of data smoothing, and discusses the particular case of Savitzky-Golay smoothing filters.

This chapter draws mathematically on the material on special functions that was presented in Chapter 6, especially §6.1–§6.4. You may wish, at this point, to review those sections.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill).

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*].

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

14.1 Moments of a Distribution: Mean, Variance, Skewness, and So Forth

When a set of values has a sufficiently strong central tendency, that is, a tendency to cluster around some particular value, then it may be useful to characterize the set by a few numbers that are related to its *moments*, the sums of integer powers of the values.

Best known is the *mean* of the values $x_1, \ldots, x_N$,

$$\overline{x} = \frac{1}{N} \sum_{j=1}^{N} x_j \tag{14.1.1}$$

which estimates the value around which central clustering occurs. Note the use of an overbar to denote the mean; angle brackets are an equally common notation, e.g., $\langle x \rangle$. You should be aware that the mean is not the only available estimator of this

quantity, nor is it necessarily the best one. For values drawn from a probability distribution with very broad "tails," the mean may converge poorly, or not at all, as the number of sampled points is increased. Alternative estimators, the *median* and the *mode*, are mentioned at the end of this section.

Having characterized a distribution's central value, one conventionally next characterizes its "width" or "variability" around that value. Here again, more than one measure is available. Most common is the *variance*,

$$\mathrm{Var}(x_1 \dots x_N) = \frac{1}{N-1} \sum_{j=1}^{N} (x_j - \overline{x})^2 \tag{14.1.2}$$

or its square root, the *standard deviation*,

$$\sigma(x_1 \dots x_N) = \sqrt{\mathrm{Var}(x_1 \dots x_N)} \tag{14.1.3}$$

Equation (14.1.2) estimates the mean squared deviation of x from its mean value. There is a long story about why the denominator of (14.1.2) is $N-1$ instead of N. If you have never heard that story, you may consult any good statistics text. Here we will be content to note that the $N-1$ *should* be changed to N if you are ever in the situation of measuring the variance of a distribution whose mean $\overline{x}$ is known *a priori* rather than being estimated from the data. (We might also comment that if the difference between N and $N-1$ ever matters to you, then you are probably up to no good anyway — e.g., trying to substantiate a questionable hypothesis with marginal data.)

As the mean depends on the first moment of the data, so do the variance and standard deviation depend on the second moment. It is not uncommon, in real life, to be dealing with a distribution whose second moment does not exist (i.e., is infinite). In this case, the variance or standard deviation is useless as a measure of the data's width around its central value: The values obtained from equations (14.1.2) or (14.1.3) will not converge with increased numbers of points, nor show any consistency from data set to data set drawn from the same distribution. This can occur even when the width of the peak looks, by eye, perfectly finite. A more robust estimator of the width is the *average deviation* or *mean absolute deviation*, defined by

$$\mathrm{ADev}(x_1 \dots x_N) = \frac{1}{N} \sum_{j=1}^{N} |x_j - \overline{x}| \tag{14.1.4}$$

One often substitutes the sample median x_{med} for $\overline{x}$ in equation (14.1.4). For any fixed sample, the median in fact minimizes the mean absolute deviation.

Statisticians have historically sniffed at the use of (14.1.4) instead of (14.1.2), since the absolute value brackets in (14.1.4) are "nonanalytic" and make theorem-proving difficult. In recent years, however, the fashion has changed, and the subject of *robust estimation* (meaning, estimation for broad distributions with significant numbers of "outlier" points) has become a popular and important one. Higher moments, or statistics involving higher powers of the input data, are almost always less robust than lower moments or statistics that involve only linear sums or (the lowest moment of all) counting.

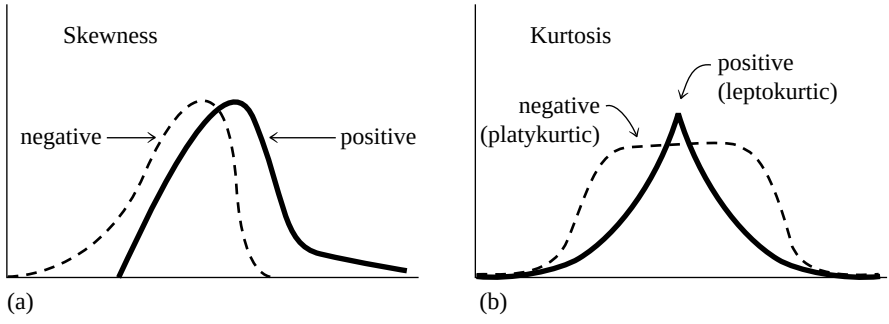


Figure 14.1.1. Distributions whose third and fourth moments are significantly different from a normal (Gaussian) distribution. (a) Skewness or third moment. (b) Kurtosis or fourth moment.

That being the case, the *skewness* or *third moment*, and the *kurtosis* or *fourth moment* should be used with caution or, better yet, not at all.

The skewness characterizes the degree of asymmetry of a distribution around its mean. While the mean, standard deviation, and average deviation are *dimensional* quantities, that is, have the same units as the measured quantities x_j, the skewness is conventionally defined in such a way as to make it *nondimensional*. It is a pure number that characterizes only the shape of the distribution. The usual definition is

$$\text{Skew}(x_1 \dots x_N) = \frac{1}{N} \sum_{j=1}^{N} \left[\frac{x_j - \overline{x}}{\sigma} \right]^3 \tag{14.1.5}$$

where $\sigma = \sigma(x_1 \dots x_N)$ is the distribution's standard deviation (14.1.3). A positive value of skewness signifies a distribution with an asymmetric tail extending out towards more positive x; a negative value signifies a distribution whose tail extends out towards more negative x (see Figure 14.1.1).

Of course, any set of N measured values is likely to give a nonzero value for (14.1.5), even if the underlying distribution is in fact symmetrical (has zero skewness). For (14.1.5) to be meaningful, we need to have some idea of *its* standard deviation as an estimator of the skewness of the underlying distribution. Unfortunately, that depends on the shape of the underlying distribution, and rather critically on its tails! For the idealized case of a normal (Gaussian) distribution, the standard deviation of (14.1.5) is approximately $\sqrt{15/N}$. In real life it is good practice to believe in skewnesses only when they are several or many times as large as this.

The kurtosis is also a nondimensional quantity. It measures the relative peakedness or flatness of a distribution. Relative to what? A normal distribution, what else! A distribution with positive kurtosis is termed *leptokurtic*; the outline of the Matterhorn is an example. A distribution with negative kurtosis is termed *platykurtic*; the outline of a loaf of bread is an example. (See Figure 14.1.1.) And, as you no doubt expect, an in-between distribution is termed *mesokurtic*.

The conventional definition of the kurtosis is

$$\text{Kurt}(x_1 \dots x_N) = \left\{ \frac{1}{N} \sum_{j=1}^{N} \left[\frac{x_j - \overline{x}}{\sigma} \right]^4 \right\} - 3 \tag{14.1.6}$$

where the -3 term makes the value zero for a normal distribution.

The standard deviation of (14.1.6) as an estimator of the kurtosis of an underlying normal distribution is $\sqrt{96/N}$. However, the kurtosis depends on such a high moment that there are many real-life distributions for which the standard deviation of (14.1.6) as an estimator is effectively infinite.

Calculation of the quantities defined in this section is perfectly straightforward. Many textbooks use the binomial theorem to expand out the definitions into sums of various powers of the data, e.g., the familiar

$$\text{Var}(x_1 \dots x_N) = \frac{1}{N-1}\left[\left(\sum_{j=1}^{N} x_j^2\right) - N\overline{x}^2\right] \approx \overline{x^2} - \overline{x}^2 \tag{14.1.7}$$

but this can magnify the roundoff error by a large factor and is generally unjustifiable in terms of computing speed. A clever way to minimize roundoff error, especially for large samples, is to use the *corrected two-pass algorithm* [1]: First calculate $\overline{x}$, then calculate $\text{Var}(x_1 \dots x_N)$ by

$$\text{Var}(x_1 \dots x_N) = \frac{1}{N-1}\left\{\sum_{j=1}^{N}(x_j - \overline{x})^2 - \frac{1}{N}\left[\sum_{j=1}^{N}(x_j - \overline{x})\right]^2\right\} \tag{14.1.8}$$

The second sum would be zero if $\overline{x}$ were exact, but otherwise it does a good job of correcting the roundoff error in the first term.

```
SUBROUTINE moment(data,n,ave,adev,sdev,var,skew,curt)
INTEGER n
REAL adev,ave,curt,sdev,skew,var,data(n)
    Given an array of data(1:n), this routine returns its mean ave, average deviation adev,
    standard deviation sdev, variance var, skewness skew, and kurtosis curt.
INTEGER j
REAL p,s,ep
if(n.le.1)pause 'n must be at least 2 in moment'
s=0.                        First pass to get the mean.
do 11 j=1,n
    s=s+data(j)
enddo 11
ave=s/n
adev=0.                     Second pass to get the first (absolute), second, third, and fourth
var=0.                          moments of the deviation from the mean.
skew=0.
curt=0.
ep=0.
do 12 j=1,n
    s=data(j)-ave
    ep=ep+s
    adev=adev+abs(s)
    p=s*s
    var=var+p
    p=p*s
    skew=skew+p
    p=p*s
    curt=curt+p
enddo 12
adev=adev/n                 Put the pieces together according to the conventional definitions.
var=(var-ep**2/n)/(n-1)         Corrected two-pass formula.
sdev=sqrt(var)
```

```
if(var.ne.0.)then
    skew=skew/(n*sdev**3)
    curt=curt/(n*var**2)-3.
else
    pause 'no skew or kurtosis when zero variance in moment'
endif
return
END
```

Semi-Invariants

The mean and variance of independent random variables are additive: If x and y are drawn independently from two, possibly different, probability distributions, then

$$\overline{(x+y)} = \overline{x} + \overline{y} \qquad \text{Var}(x+y) = \text{Var}(x) + \text{Var}(x) \tag{14.1.9}$$

Higher moments are not, in general, additive. However, certain combinations of them, called *semi-invariants*, are in fact additive. If the centered moments of a distribution are denoted M_k,

$$M_k \equiv \left\langle (x_i - \overline{x})^k \right\rangle \tag{14.1.10}$$

so that, e.g., $M_2 = \text{Var}(x)$, then the first few semi-invariants, denoted I_k are given by

$$\begin{aligned} I_2 &= M_2 \qquad I_3 = M_3 \qquad I_4 = M_4 - 3M_2^2 \\ I_5 &= M_5 - 10M_2M_3 \qquad I_6 = M_6 - 15M_2M_4 - 10M_3^2 + 30M_2^3 \end{aligned} \tag{14.1.11}$$

Notice that the skewness and kurtosis, equations (14.1.5) and (14.1.6) are simple powers of the semi-invariants,

$$\text{Skew}(x) = I_3/I_2^{3/2} \qquad \text{Kurt}(x) = I_4/I_2^2 \tag{14.1.12}$$

A Gaussian distribution has all its semi-invariants higher than I_2 equal to zero. A Poisson distribution has all of its semi-invariants equal to its mean. For more details, see [2].

Median and Mode

The median of a probability distribution function $p(x)$ is the value x_{med} for which larger and smaller values of x are equally probable:

$$\int_{-\infty}^{x_{\text{med}}} p(x)\, dx = \frac{1}{2} = \int_{x_{\text{med}}}^{\infty} p(x)\, dx \tag{14.1.13}$$

The median of a distribution is estimated from a sample of values $x_1, \ldots, x_N$ by finding that value x_i which has equal numbers of values above it and below it. Of course, this is not possible when N is even. In that case it is conventional to estimate the median as the mean of the unique *two* central values. If the values x_j $j = 1, \ldots, N$ are sorted into ascending (or, for that matter, descending) order, then the formula for the median is

$$x_{\text{med}} = \begin{cases} x_{(N+1)/2}, & N \text{ odd} \\ \frac{1}{2}(x_{N/2} + x_{(N/2)+1}), & N \text{ even} \end{cases} \tag{14.1.14}$$

If a distribution has a strong central tendency, so that most of its area is under a single peak, then the median is an estimator of the central value. It is a more robust estimator than the mean is: The median fails as an estimator only if the area in the tails is large, while the mean fails if the first moment of the tails is large; it is easy to construct examples where the first moment of the tails is large even though their area is negligible.

To find the median of a set of values, one can proceed by sorting the set and then applying (14.1.14). This is a process of order $N \log N$. You might rightly think that this is wasteful, since it yields much more information than just the median (e.g., the upper and lower quartile points, the deciles, etc.). In fact, we saw in §8.5 that the element $x_{(N+1)/2}$ can be located in of order N operations. Consult that section for routines.

The *mode* of a probability distribution function $p(x)$ is the value of x where it takes on a maximum value. The mode is useful primarily when there is a single, sharp maximum, in which case it estimates the central value. Occasionally, a distribution will be *bimodal*, with two relative maxima; then one may wish to know the two modes individually. Note that, in such cases, the mean and median are not very useful, since they will give only a "compromise" value between the two peaks.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 2.

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*], vol. 1, §10.15

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Chan, T.F., Golub, G.H., and LeVeque, R.J. 1983, *American Statistician*, vol. 37, pp. 242–247. [1]

Cramér, H. 1946, *Mathematical Methods of Statistics* (Princeton: Princeton University Press), §15.10. [2]

14.2 Do Two Distributions Have the Same Means or Variances?

Not uncommonly we want to know whether two distributions have the same mean. For example, a first set of measured values may have been gathered before some event, a second set after it. We want to know whether the event, a "treatment" or a "change in a control parameter," made a difference.

Our first thought is to ask "how many standard deviations" one sample mean is from the other. That number may in fact be a useful thing to know. It does relate to the strength or "importance" of a difference of means *if that difference is genuine*. However, by itself, it says nothing about whether the difference *is* genuine, that is, statistically significant. A difference of means can be very small compared to the standard deviation, and yet very significant, if the number of data points is large. Conversely, a difference may be moderately large but not significant, if the data

If a distribution has a strong central tendency, so that most of its area is under a single peak, then the median is an estimator of the central value. It is a more robust estimator than the mean is: The median fails as an estimator only if the area in the tails is large, while the mean fails if the first moment of the tails is large; it is easy to construct examples where the first moment of the tails is large even though their area is negligible.

To find the median of a set of values, one can proceed by sorting the set and then applying (14.1.14). This is a process of order $N \log N$. You might rightly think that this is wasteful, since it yields much more information than just the median (e.g., the upper and lower quartile points, the deciles, etc.). In fact, we saw in §8.5 that the element $x_{(N+1)/2}$ can be located in of order N operations. Consult that section for routines.

The *mode* of a probability distribution function $p(x)$ is the value of x where it takes on a maximum value. The mode is useful primarily when there is a single, sharp maximum, in which case it estimates the central value. Occasionally, a distribution will be *bimodal*, with two relative maxima; then one may wish to know the two modes individually. Note that, in such cases, the mean and median are not very useful, since they will give only a "compromise" value between the two peaks.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 2.

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*], vol. 1, §10.15

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Chan, T.F., Golub, G.H., and LeVeque, R.J. 1983, *American Statistician*, vol. 37, pp. 242–247. [1]

Cramér, H. 1946, *Mathematical Methods of Statistics* (Princeton: Princeton University Press), §15.10. [2]

14.2 Do Two Distributions Have the Same Means or Variances?

Not uncommonly we want to know whether two distributions have the same mean. For example, a first set of measured values may have been gathered before some event, a second set after it. We want to know whether the event, a "treatment" or a "change in a control parameter," made a difference.

Our first thought is to ask "how many standard deviations" one sample mean is from the other. That number may in fact be a useful thing to know. It does relate to the strength or "importance" of a difference of means *if that difference is genuine*. However, by itself, it says nothing about whether the difference *is* genuine, that is, statistically significant. A difference of means can be very small compared to the standard deviation, and yet very significant, if the number of data points is large. Conversely, a difference may be moderately large but not significant, if the data

are sparse. We will be meeting these distinct concepts of *strength* and *significance* several times in the next few sections.

A quantity that measures the significance of a difference of means is not the number of standard deviations that they are apart, but the number of so-called *standard errors* that they are apart. The standard error of a set of values measures the accuracy with which the sample mean estimates the population (or "true") mean. Typically the standard error is equal to the sample's standard deviation divided by the square root of the number of points in the sample.

Student's t-test for Significantly Different Means

Applying the concept of standard error, the conventional statistic for measuring the significance of a difference of means is termed *Student's t*. When the two distributions are thought to have the same variance, but possibly different means, then Student's t is computed as follows: First, estimate the standard error of the difference of the means, s_D, from the "pooled variance" by the formula

$$s_D = \sqrt{\frac{\sum_{i\in A}(x_i - \overline{x_A})^2 + \sum_{i\in B}(x_i - \overline{x_B})^2}{N_A + N_B - 2}\left(\frac{1}{N_A} + \frac{1}{N_B}\right)} \tag{14.2.1}$$

where each sum is over the points in one sample, the first or second, each mean likewise refers to one sample or the other, and N_A and N_B are the numbers of points in the first and second samples, respectively. Second, compute t by

$$t = \frac{\overline{x_A} - \overline{x_B}}{s_D} \tag{14.2.2}$$

Third, evaluate the significance of this value of t for Student's distribution with $N_A + N_B - 2$ degrees of freedom, by equations (6.4.7) and (6.4.9), and by the routine `betai` (incomplete beta function) of §6.4.

The significance is a number between zero and one, and is the probability that $|t|$ could be this large or larger just by chance, for distributions with equal means. Therefore, a small numerical value of the significance (0.05 or 0.01) means that the observed difference is "very significant." The function $A(t|\nu)$ in equation (6.4.7) is one minus the significance.

As a routine, we have

```
      SUBROUTINE ttest(data1,n1,data2,n2,t,prob)
      INTEGER n1,n2
      REAL prob,t,data1(n1),data2(n2)
C     USES avevar,betai
        Given the arrays data1(1:n1) and data2(1:n2), this routine returns Student's t as t,
        and its significance as prob, small values of prob indicating that the arrays have significantly
        different means. The data arrays are assumed to be drawn from populations with the same
        true variance.
      REAL ave1,ave2,df,var,var1,var2,betai
      call avevar(data1,n1,ave1,var1)
      call avevar(data2,n2,ave2,var2)
      df=n1+n2-2                                Degrees of freedom.
      var=((n1-1)*var1+(n2-1)*var2)/df          Pooled variance.
      t=(ave1-ave2)/sqrt(var*(1./n1+1./n2))
      prob=betai(0.5*df,0.5,df/(df+t**2))       See equation (6.4.9).
      return
      END
```

which makes use of the following routine for computing the mean and variance of a set of numbers,

```
SUBROUTINE avevar(data,n,ave,var)
INTEGER n
REAL ave,var,data(n)
    Given array data(1:n), returns its mean as ave and its variance as var.
INTEGER j
REAL s,ep
ave=0.0
do 11 j=1,n
    ave=ave+data(j)
enddo 11
ave=ave/n
var=0.0
ep=0.0
do 12 j=1,n
    s=data(j)-ave
    ep=ep+s
    var=var+s*s
enddo 12
var=(var-ep**2/n)/(n-1)          Corrected two-pass formula (14.1.8).
return
END
```

The next case to consider is where the two distributions have significantly different variances, but we nevertheless want to know if their means are the same or different. (A treatment for baldness has caused some patients to *lose* all their hair and turned others into werewolves, but we want to know if it helps cure baldness *on the average*!) Be suspicious of the unequal-variance t-test: If two distributions have very different variances, then they may also be substantially different in shape; in that case, the difference of the means may not be a particularly useful thing to know.

To find out whether the two data sets have variances that are significantly different, you use the *F-test*, described later on in this section.

The relevant statistic for the unequal variance t-test is

$$t = \frac{\overline{x_A} - \overline{x_B}}{[\mathrm{Var}(x_A)/N_A + \mathrm{Var}(x_B)/N_B]^{1/2}} \tag{14.2.3}$$

This statistic is distributed *approximately* as Student's t with a number of degrees of freedom equal to

$$\frac{\left[\dfrac{\mathrm{Var}(x_A)}{N_A} + \dfrac{\mathrm{Var}(x_B)}{N_B}\right]^2}{\dfrac{[\mathrm{Var}(x_A)/N_A]^2}{N_A - 1} + \dfrac{[\mathrm{Var}(x_B)/N_B]^2}{N_B - 1}} \tag{14.2.4}$$

Expression (14.2.4) is in general not an integer, but equation (6.4.7) doesn't care.

The routine is

```
SUBROUTINE tutest(data1,n1,data2,n2,t,prob)
INTEGER n1,n2
REAL prob,t,data1(n1),data2(n2)
C  USES avevar,betai
    Given the arrays data1(1:n1) and data2(1:n2), this routine returns Student's t as t,
    and its significance as prob, small values of prob indicating that the arrays have significantly
```

different means. The data arrays are allowed to be drawn from populations with unequal variances.

```
REAL ave1,ave2,df,var1,var2,betai
call avevar(data1,n1,ave1,var1)
call avevar(data2,n2,ave2,var2)
t=(ave1-ave2)/sqrt(var1/n1+var2/n2)
df=(var1/n1+var2/n2)**2/((var1/n1)**2/(n1-1)+(var2/n2)**2/(n2-1))
prob=betai(0.5*df,0.5,df/(df+t**2))
return
END
```

Our final example of a Student's t test is the case of *paired samples*. Here we imagine that much of the variance in *both* samples is due to effects that are point-by-point identical in the two samples. For example, we might have two job candidates who have each been rated by the same ten members of a hiring committee. We want to know if the means of the ten scores differ significantly. We first try `ttest` above, and obtain a value of `prob` that is not especially significant (e.g., > 0.05). But perhaps the significance is being washed out by the tendency of some committee members always to give high scores, others always to give low scores, which increases the apparent variance and thus decreases the significance of any difference in the means. We thus try the paired-sample formulas,

$$\mathrm{Cov}(x_A, x_B) \equiv \frac{1}{N-1}\sum_{i=1}^{N}(x_{Ai} - \overline{x_A})(x_{Bi} - \overline{x_B}) \tag{14.2.5}$$

$$s_D = \left[\frac{\mathrm{Var}(x_A) + \mathrm{Var}(x_B) - 2\mathrm{Cov}(x_A, x_B)}{N}\right]^{1/2} \tag{14.2.6}$$

$$t = \frac{\overline{x_A} - \overline{x_B}}{s_D} \tag{14.2.7}$$

where N is the number in each sample (number of pairs). Notice that it is important that a particular value of i label the corresponding points in each sample, that is, the ones that are paired. The significance of the t statistic in (14.2.7) is evaluated for $N - 1$ degrees of freedom.

The routine is

```
      SUBROUTINE tptest(data1,data2,n,t,prob)
      INTEGER n
      REAL prob,t,data1(n),data2(n)
C     USES avevar,betai
        Given the paired arrays data1(1:n) and data2(1:n), this routine returns Student's t for
        paired data as t, and its significance as prob, small values of prob indicating a significant
        difference of means.
      INTEGER j
      REAL ave1,ave2,cov,df,sd,var1,var2,betai
      call avevar(data1,n,ave1,var1)
      call avevar(data2,n,ave2,var2)
      cov=0.
      do 11 j=1,n
          cov=cov+(data1(j)-ave1)*(data2(j)-ave2)
      enddo 11
      df=n-1
      cov=cov/df
      sd=sqrt((var1+var2-2.*cov)/n)
      t=(ave1-ave2)/sd
```

```
prob=betai(0.5*df,0.5,df/(df+t**2))
return
END
```

F-Test for Significantly Different Variances

The *F-test* tests the hypothesis that two samples have different variances by trying to reject the null hypothesis that their variances are actually consistent. The statistic F is the ratio of one variance to the other, so values either $\gg 1$ or $\ll 1$ will indicate very significant differences. The distribution of F in the null case is given in equation (6.4.11), which is evaluated using the routine `betai`. In the most common case, we are willing to disprove the null hypothesis (of equal variances) by either very large or very small values of F, so the correct significance is *two-tailed*, the sum of two incomplete beta functions. It turns out, by equation (6.4.3), that the two tails are always equal; we need compute only one, and double it. Occasionally, when the null hypothesis is strongly viable, the identity of the two tails can become confused, giving an indicated probability greater than one. Changing the probability to two minus itself correctly exchanges the tails. These considerations and equation (6.4.3) give the routine

```
      SUBROUTINE ftest(data1,n1,data2,n2,f,prob)
      INTEGER n1,n2
      REAL f,prob,data1(n1),data2(n2)
C     USES avevar,betai
        Given the arrays data1(1:n1) and data2(1:n2), this routine returns the value of f, and
        its significance as prob. Small values of prob indicate that the two arrays have significantly
        different variances.
      REAL ave1,ave2,df1,df2,var1,var2,betai
      call avevar(data1,n1,ave1,var1)
      call avevar(data2,n2,ave2,var2)
      if(var1.gt.var2)then          Make F the ratio of the larger variance to the smaller one.
          f=var1/var2
          df1=n1-1
          df2=n2-1
      else
          f=var2/var1
          df1=n2-1
          df2=n1-1
      endif
      prob=2.*betai(0.5*df2,0.5*df1,df2/(df2+df1*f))
      if(prob.gt.1.)prob=2.-prob
      return
      END
```

CITED REFERENCES AND FURTHER READING:

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapter IX(B).

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

14.3 Are Two Distributions Different?

Given two sets of data, we can generalize the questions asked in the previous section and ask the single question: Are the two sets drawn from the same distribution function, or from different distribution functions? Equivalently, in proper statistical language, "Can we disprove, to a certain required level of significance, the null hypothesis that two data sets are drawn from the same population distribution function?" Disproving the null hypothesis in effect proves that the data sets are from different distributions. Failing to disprove the null hypothesis, on the other hand, only shows that the data sets can be *consistent* with a single distribution function. One can never *prove* that two data sets come from a single distribution, since (e.g.) no practical amount of data can distinguish between two distributions which differ only by one part in 10^{10}.

Proving that two distributions are different, or showing that they are consistent, is a task that comes up all the time in many areas of research: Are the visible stars distributed uniformly in the sky? (That is, is the distribution of stars as a function of declination — position in the sky — the same as the distribution of sky area as a function of declination?) Are educational patterns the same in Brooklyn as in the Bronx? (That is, are the distributions of people as a function of last-grade-attended the same?) Do two brands of fluorescent lights have the same distribution of burn-out times? Is the incidence of chicken pox the same for first-born, second-born, third-born children, etc.?

These four examples illustrate the four combinations arising from two different dichotomies: (1) The data are either continuous or binned. (2) Either we wish to compare one data set to a known distribution, or we wish to compare two equally unknown data sets. The data sets on fluorescent lights and on stars are continuous, since we can be given lists of individual burnout times or of stellar positions. The data sets on chicken pox and educational level are binned, since we are given tables of numbers of events in discrete categories: first-born, second-born, etc.; or 6th Grade, 7th Grade, etc. Stars and chicken pox, on the other hand, share the property that the null hypothesis is a known distribution (distribution of area in the sky, or incidence of chicken pox in the general population). Fluorescent lights and educational level involve the comparison of two equally unknown data sets (the two brands, or Brooklyn and the Bronx).

One can always turn continuous data into binned data, by grouping the events into specified ranges of the continuous variable(s): declinations between 0 and 10 degrees, 10 and 20, 20 and 30, etc. Binning involves a loss of information, however. Also, there is often considerable arbitrariness as to how the bins should be chosen. Along with many other investigators, we prefer to avoid unnecessary binning of data.

The accepted test for differences between binned distributions is the *chi-square test*. For continuous data as a function of a single variable, the most generally accepted test is the *Kolmogorov-Smirnov test*. We consider each in turn.

Chi-Square Test

Suppose that N_i is the number of events observed in the ith bin, and that n_i is the number expected according to some known distribution. Note that the N_i's are

14.3 Are Two Distributions Different?

Given two sets of data, we can generalize the questions asked in the previous section and ask the single question: Are the two sets drawn from the same distribution function, or from different distribution functions? Equivalently, in proper statistical language, "Can we disprove, to a certain required level of significance, the null hypothesis that two data sets are drawn from the same population distribution function?" Disproving the null hypothesis in effect proves that the data sets are from different distributions. Failing to disprove the null hypothesis, on the other hand, only shows that the data sets can be *consistent* with a single distribution function. One can never *prove* that two data sets come from a single distribution, since (e.g.) no practical amount of data can distinguish between two distributions which differ only by one part in 10^{10}.

Proving that two distributions are different, or showing that they are consistent, is a task that comes up all the time in many areas of research: Are the visible stars distributed uniformly in the sky? (That is, is the distribution of stars as a function of declination — position in the sky — the same as the distribution of sky area as a function of declination?) Are educational patterns the same in Brooklyn as in the Bronx? (That is, are the distributions of people as a function of last-grade-attended the same?) Do two brands of fluorescent lights have the same distribution of burn-out times? Is the incidence of chicken pox the same for first-born, second-born, third-born children, etc.?

These four examples illustrate the four combinations arising from two different dichotomies: (1) The data are either continuous or binned. (2) Either we wish to compare one data set to a known distribution, or we wish to compare two equally unknown data sets. The data sets on fluorescent lights and on stars are continuous, since we can be given lists of individual burnout times or of stellar positions. The data sets on chicken pox and educational level are binned, since we are given tables of numbers of events in discrete categories: first-born, second-born, etc.; or 6th Grade, 7th Grade, etc. Stars and chicken pox, on the other hand, share the property that the null hypothesis is a known distribution (distribution of area in the sky, or incidence of chicken pox in the general population). Fluorescent lights and educational level involve the comparison of two equally unknown data sets (the two brands, or Brooklyn and the Bronx).

One can always turn continuous data into binned data, by grouping the events into specified ranges of the continuous variable(s): declinations between 0 and 10 degrees, 10 and 20, 20 and 30, etc. Binning involves a loss of information, however. Also, there is often considerable arbitrariness as to how the bins should be chosen. Along with many other investigators, we prefer to avoid unnecessary binning of data.

The accepted test for differences between binned distributions is the *chi-square test*. For continuous data as a function of a single variable, the most generally accepted test is the *Kolmogorov-Smirnov test*. We consider each in turn.

Chi-Square Test

Suppose that N_i is the number of events observed in the ith bin, and that n_i is the number expected according to some known distribution. Note that the N_i's are

integers, while the n_i's may not be. Then the chi-square statistic is

$$\chi^2 = \sum_i \frac{(N_i - n_i)^2}{n_i} \tag{14.3.1}$$

where the sum is over all bins. A large value of χ^2 indicates that the null hypothesis (that the N_i's are drawn from the population represented by the n_i's) is rather unlikely.

Any term j in (14.3.1) with $0 = n_j = N_j$ should be omitted from the sum. A term with $n_j = 0,\ N_j \neq 0$ gives an infinite χ^2, as it should, since in this case the N_i's cannot possibly be drawn from the n_i's!

The *chi-square probability function* $Q(\chi^2|\nu)$ is an incomplete gamma function, and was already discussed in §6.2 (see equation 6.2.18). Strictly speaking $Q(\chi^2|\nu)$ is the probability that the sum of the squares of ν random *normal* variables of unit variance (and zero mean) will be greater than χ^2. The terms in the sum (14.3.1) are not individually normal. However, if either the number of bins is large ($\gg 1$), or the number of events in each bin is large ($\gg 1$), then the chi-square probability function is a good approximation to the distribution of (14.3.1) in the case of the null hypothesis. Its use to estimate the significance of the chi-square test is standard.

The appropriate value of ν, the number of degrees of freedom, bears some additional discussion. If the data are collected with the model n_i's fixed — that is, not later renormalized to fit the total observed number of events ΣN_i — then ν equals the number of bins N_B. (Note that this is *not* the total number of *events*!) Much more commonly, the n_i's are normalized after the fact so that their sum equals the sum of the N_i's. In this case the correct value for ν is $N_B - 1$, and the model is said to have one constraint (`knstrn=1` in the program below). If the model that gives the n_i's has additional free parameters that were adjusted after the fact to agree with the data, then each of these additional "fitted" parameters decreases ν (and increases `knstrn`) by one additional unit.

We have, then, the following program:

```
      SUBROUTINE chsone(bins,ebins,nbins,knstrn,df,chsq,prob)
      INTEGER knstrn,nbins
      REAL chsq,df,prob,bins(nbins),ebins(nbins)
C     USES gammq
        Given the array bins(1:nbins) containing the observed numbers of events, and an array
        ebins(1:nbins) containing the expected numbers of events, and given the number of
        constraints knstrn (normally one), this routine returns (trivially) the number of degrees of
        freedom df, and (nontrivially) the chi-square chsq and the significance prob. A small value
        of prob indicates a significant difference between the distributions bins and ebins. Note
        that bins and ebins are both real arrays, although bins will normally contain integer
        values.
      INTEGER j
      REAL gammq
      df=nbins-knstrn
      chsq=0.
      do 11 j=1,nbins
        if(ebins(j).le.0.)pause 'bad expected number in chsone'
        chsq=chsq+(bins(j)-ebins(j))**2/ebins(j)
      enddo 11
      prob=gammq(0.5*df,0.5*chsq)          Chi-square probability function. See §6.2.
      return
      END
```

Next we consider the case of comparing *two* binned data sets. Let R_i be the number of events in bin i for the first data set, S_i the number of events in the same bin i for the second data set. Then the chi-square statistic is

$$\chi^2 = \sum_i \frac{(R_i - S_i)^2}{R_i + S_i} \tag{14.3.2}$$

Comparing (14.3.2) to (14.3.1), you should note that the denominator of (14.3.2) is *not* just the average of R_i and S_i (which would be an estimator of n_i in 14.3.1). Rather, it is twice the average, the sum. The reason is that each term in a chi-square sum is supposed to approximate the square of a normally distributed quantity with unit variance. The variance of the difference of two normal quantities is the sum of their individual variances, not the average.

If the data were collected in such a way that the sum of the R_i's is necessarily equal to the sum of S_i's, then the number of degrees of freedom is equal to one less than the number of bins, $N_B - 1$ (that is, `knstrn` $= 1$), the usual case. If this requirement were absent, then the number of degrees of freedom would be N_B. Example: A birdwatcher wants to know whether the distribution of sighted birds as a function of species is the same this year as last. Each bin corresponds to one species. If the birdwatcher takes his data to be the first 1000 birds that he saw in each year, then the number of degrees of freedom is $N_B - 1$. If he takes his data to be all the birds he saw on a random sample of days, the same days in each year, then the number of degrees of freedom is N_B (`knstrn` $= 0$). In this latter case, note that he is also testing whether the birds were more numerous overall in one year or the other: That is the extra degree of freedom. Of course, any additional constraints on the data set lower the number of degrees of freedom (i.e., increase `knstrn` to *more positive* values) in accordance with their number.

The program is

```
      SUBROUTINE chstwo(bins1,bins2,nbins,knstrn,df,chsq,prob)
      INTEGER knstrn,nbins
      REAL chsq,df,prob,bins1(nbins),bins2(nbins)
C     USES gammq
        Given the arrays bins1(1:nbins) and bins2(1:nbins), containing two sets of binned
        data, and given the number of constraints knstrn (normally 1 or 0), this routine returns
        the number of degrees of freedom df, the chi-square chsq, and the significance prob.
        A small value of prob indicates a significant difference between the distributions bins1
        and bins2. Note that bins1 and bins2 are both real arrays, although they will normally
        contain integer values.
      INTEGER j
      REAL gammq
      df=nbins-knstrn
      chsq=0.
      do 11 j=1,nbins
        if(bins1(j).eq.0..and.bins2(j).eq.0.)then
          df=df-1.                    No data means one less degree of freedom.
        else
          chsq=chsq+(bins1(j)-bins2(j))**2/(bins1(j)+bins2(j))
        endif
      enddo 11
      prob=gammq(0.5*df,0.5*chsq)     Chi-square probability function. See §6.2.
      return
      END
```

Equation (14.3.2) and the routine `chstwo` both apply to the case where the total number of data points is the same in the two binned sets. For unequal numbers of data points, the formula analogous to (14.3.2) is

$$\chi^2 = \sum_i \frac{(\sqrt{S/R}R_i - \sqrt{R/S}S_i)^2}{R_i + S_i} \tag{14.3.3}$$

where

$$R \equiv \sum_i R_i \qquad S \equiv \sum_i S_i \tag{14.3.4}$$

are the respective numbers of data points. It is straightforward to make the corresponding change in `chstwo`.

Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (or *K–S*) test is applicable to unbinned distributions that are functions of a single independent variable, that is, to data sets where each data point can be associated with a single number (lifetime of each lightbulb when it burns out, or declination of each star). In such cases, the list of data points can be easily converted to an unbiased estimator $S_N(x)$ of the *cumulative* distribution function of the probability distribution from which it was drawn: If the N events are located at values $x_i,\ i = 1, \ldots, N$, then $S_N(x)$ is the function giving the fraction of data points to the left of a given value x. This function is obviously constant between consecutive (i.e., sorted into ascending order) x_i's, and jumps by the same constant $1/N$ at each x_i. (See Figure 14.3.1.)

Different distribution functions, or sets of data, give different cumulative distribution function estimates by the above procedure. However, all cumulative distribution functions agree at the smallest allowable value of x (where they are zero), and at the largest allowable value of x (where they are unity). (The smallest and largest values might of course be $\pm\infty$.) So it is the behavior between the largest and smallest values that distinguishes distributions.

One can think of any number of statistics to measure the overall difference between two cumulative distribution functions: the absolute value of the area between them, for example. Or their integrated mean square difference. The Kolmogorov-Smirnov D is a particularly simple measure: It is defined as the *maximum value* of the absolute difference between two cumulative distribution functions. Thus, for comparing one data set's $S_N(x)$ to a known cumulative distribution function $P(x)$, the K–S statistic is

$$D = \max_{-\infty<x<\infty} |S_N(x) - P(x)| \tag{14.3.5}$$

while for comparing two different cumulative distribution functions $S_{N_1}(x)$ and $S_{N_2}(x)$, the K–S statistic is

$$D = \max_{-\infty<x<\infty} |S_{N_1}(x) - S_{N_2}(x)| \tag{14.3.6}$$

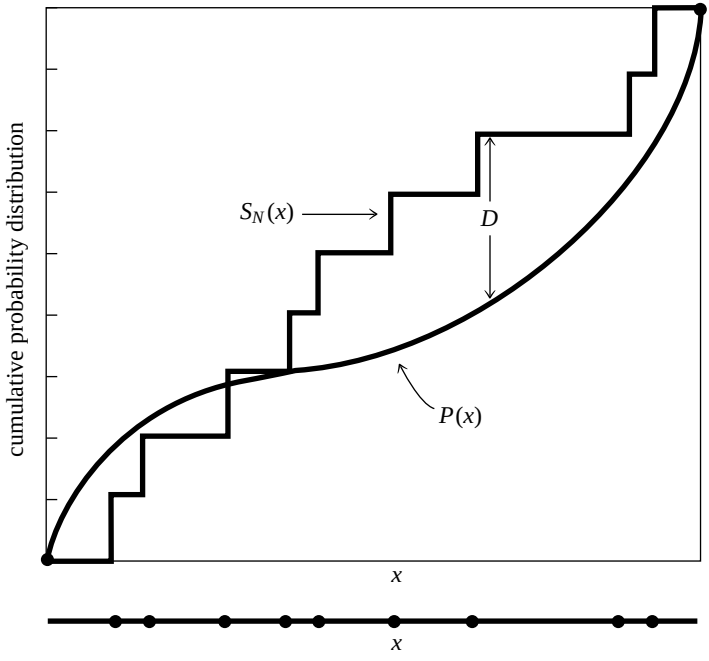


Figure 14.3.1. Kolmogorov-Smirnov statistic D. A measured distribution of values in x (shown as N dots on the lower abscissa) is to be compared with a theoretical distribution whose cumulative probability distribution is plotted as $P(x)$. A step-function cumulative probability distribution $S_N(x)$ is constructed, one that rises an equal amount at each measured point. D is the greatest distance between the two cumulative distributions.

What makes the K–S statistic useful is that *its* distribution in the case of the null hypothesis (data sets drawn from the same distribution) can be calculated, at least to useful approximation, thus giving the significance of any observed nonzero value of D. A central feature of the K–S test is that it is invariant under reparametrization of x; in other words, you can locally slide or stretch the x axis in Figure 14.3.1, and the maximum distance D remains unchanged. For example, you will get the same significance using x as using $\log x$.

The function that enters into the calculation of the significance can be written as the following sum:

$$Q_{KS}(\lambda) = 2\sum_{j=1}^{\infty}(-1)^{j-1}\, e^{-2j^2\lambda^2} \tag{14.3.7}$$

which is a monotonic function with the limiting values

$$Q_{KS}(0) = 1 \qquad Q_{KS}(\infty) = 0 \tag{14.3.8}$$

In terms of this function, the significance level of an observed value of D (as a disproof of the null hypothesis that the distributions are the same) is given approximately [1] by the formula

$$\text{Probability}\,(D > \text{observed}\,) = Q_{KS}\Big(\Big[\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}\Big]\; D\Big) \tag{14.3.9}$$

where N_e is the effective number of data points, $N_e = N$ for the case (14.3.5) of one distribution, and

$$N_e = \frac{N_1 N_2}{N_1 + N_2} \tag{14.3.10}$$

for the case (14.3.6) of two distributions, where N_1 is the number of data points in the first distribution, N_2 the number in the second.

The nature of the approximation involved in (14.3.9) is that it becomes asymptotically accurate as the N_e becomes large, but is already quite good for $N_e \geq 4$, as small a number as one might ever actually use. (See [1].)

So, we have the following routines for the cases of one and two distributions:

```
      SUBROUTINE ksone(data,n,func,d,prob)
      INTEGER n
      REAL d,data(n),func,prob
      EXTERNAL func
C     USES probks,sort
        Given an array data(1:n), and given a user-supplied function of a single variable func
        which is a cumulative distribution function ranging from 0 (for smallest values of its argu-
        ment) to 1 (for largest values of its argument), this routine returns the K–S statistic d, and
        the significance level prob. Small values of prob show that the cumulative distribution
        function of data is significantly different from func. The array data is modified by being
        sorted into ascending order.
      INTEGER j
      REAL dt,en,ff,fn,fo,probks
      call sort(n,data)                    If the data are already sorted into ascending or-
      en=n                                     der, then this call can be omitted.
      d=0.
      fo=0.                                Data's c.d.f. before the next step.
      do 11 j=1,n                          Loop over the sorted data points.
        fn=j/en                            Data's c.d.f. after this step.
        ff=func(data(j))                   Compare to the user-supplied function.
        dt=max(abs(fo-ff),abs(fn-ff))      Maximum distance.
        if(dt.gt.d)d=dt
        fo=fn
      enddo 11
      en=sqrt(en)
      prob=probks((en+0.12+0.11/en)*d)     Compute significance.
      return
      END


      SUBROUTINE kstwo(data1,n1,data2,n2,d,prob)
      INTEGER n1,n2
      REAL d,prob,data1(n1),data2(n2)
C     USES probks,sort
        Given an array data1(1:n1), and an array data2(1:n2), this routine returns the K–
        S statistic d, and the significance level prob for the null hypothesis that the data sets
        are drawn from the same distribution. Small values of prob show that the cumulative
        distribution function of data1 is significantly different from that of data2. The arrays
        data1 and data2 are modified by being sorted into ascending order.
      INTEGER j1,j2
      REAL d1,d2,dt,en1,en2,en,fn1,fn2,probks
      call sort(n1,data1)
      call sort(n2,data2)
      en1=n1
      en2=n2
      j1=1                                 Next value of data1 to be processed.
      j2=1                                 Ditto, data2.
```

```
      fn1=0.
      fn2=0.
      d=0.
1     if(j1.le.n1.and.j2.le.n2)then          If we are not done...
          d1=data1(j1)
          d2=data2(j2)
          if(d1.le.d2)then                   Next step is in data1.
              fn1=j1/en1
              j1=j1+1
          endif
          if(d2.le.d1)then                   Next step is in data2.
              fn2=j2/en2
              j2=j2+1
          endif
          dt=abs(fn2-fn1)
          if(dt.gt.d)d=dt
      goto 1
      endif
      en=sqrt(en1*en2/(en1+en2))
      prob=probks((en+0.12+0.11/en)*d)       Compute significance.
      return
      END
```

Both of the above routines use the following routine for calculating the function Q_{KS}:

```
      FUNCTION probks(alam)
      REAL probks,alam,EPS1,EPS2
      PARAMETER (EPS1=0.001, EPS2=1.e-8)
         Kolmogorov-Smirnov probability function.
      INTEGER j
      REAL a2,fac,term,termbf
      a2=-2.*alam**2
      fac=2.
      probks=0.
      termbf=0.                      Previous term in sum.
      do 11 j=1,100
          term=fac*exp(a2*j**2)
          probks=probks+term
          if(abs(term).le.EPS1*termbf.or.abs(term).le.EPS2*probks)return
          fac=-fac                   Alternating signs in sum.
          termbf=abs(term)
      enddo 11
      probks=1.                      Get here only by failing to converge.
      return
      END
```

Variants on the K–S Test

The sensitivity of the K–S test to deviations from a cumulative distribution function $P(x)$ is not independent of x. In fact, the K–S test tends to be most sensitive around the median value, where $P(x) = 0.5$, and less sensitive at the extreme ends of the distribution, where $P(x)$ is near 0 or 1. The reason is that the difference $|S_N(x) - P(x)|$ does not, in the null hypothesis, have a probability distribution that is independent of x. Rather, its variance is proportional to $P(x)[1 - P(x)]$, which is largest at $P = 0.5$. Since the K–S statistic (14.3.5) is the maximum difference over all x of two cumulative distribution functions, a deviation that might be statistically significant at *its own* value of x gets compared to the expected chance deviation at $P = 0.5$, and is thus discounted. A result is that, while the K–S test is good at

finding *shifts* in a probability distribution, especially changes in the median value, it is not always so good at finding *spreads*, which more affect the tails of the probability distribution, and which may leave the median unchanged.

One way of increasing the power of the K–S statistic out on the tails is to replace D (equation 14.3.5) by a so-called *stabilized* or *weighted* statistic [2-4], for example the *Anderson-Darling statistic*,

$$D^* = \max_{-\infty<x<\infty} \frac{|S_N(x) - P(x)|}{\sqrt{P(x)[1-P(x)]}} \tag{14.3.11}$$

Unfortunately, there is no simple formula analogous to equations (14.3.7) and (14.3.9) for this statistic, although Noé [5] gives a computational method using a recursion relation and provides a graph of numerical results. There are many other possible similar statistics, for example

$$D^{**} = \int_{P=0}^{1} \frac{|S_N(x) - P(x)|}{\sqrt{P(x)[1-P(x)]}} dP(x) \tag{14.3.12}$$

which is also discussed by Anderson and Darling (see [3]).

Another approach, which we prefer as simpler and more direct, is due to Kuiper [6,7]. We already mentioned that the standard K–S test is invariant under reparametrizations of the variable x. An even more general symmetry, which guarantees equal sensitivities at all values of x, is to wrap the x axis around into a circle (identifying the points at $\pm\infty$), and to look for a statistic that is now invariant under all shifts and parametrizations on the circle. This allows, for example, a probability distribution to be "cut" at some central value of x, and the left and right halves to be interchanged, without altering the statistic or its significance.

Kuiper's statistic, defined as

$$V = D_+ + D_- = \max_{-\infty<x<\infty} [S_N(x) - P(x)] + \max_{-\infty<x<\infty} [P(x) - S_N(x)] \tag{14.3.13}$$

is the sum of the maximum distance of $S_N(x)$ *above and below* $P(x)$. You should be able to convince yourself that this statistic has the desired invariance on the circle: Sketch the indefinite integral of two probability distributions defined on the circle as a function of angle around the circle, as the angle goes through several times 360°. If you change the starting point of the integration, D_+ and D_- change individually, but their sum is constant.

Furthermore, there is a simple formula for the asymptotic distribution of the statistic V, directly analogous to equations (14.3.7)–(14.3.10). Let

$$Q_{KP}(\lambda) = 2\sum_{j=1}^{\infty} (4j^2\lambda^2 - 1)e^{-2j^2\lambda^2} \tag{14.3.14}$$

which is monotonic and satisfies

$$Q_{KP}(0) = 1 \qquad Q_{KP}(\infty) = 0 \tag{14.3.15}$$

In terms of this function the significance level is [1]

$$\text{Probability } (V > \text{observed }) = Q_{KP}\left(\left[\sqrt{N_e} + 0.155 + 0.24/\sqrt{N_e}\right] D\right) \tag{14.3.16}$$

Here N_e is N in the one-sample case, or is given by equation (14.3.10) in the case of two samples.

Of course, Kuiper's test is ideal for any problem originally defined on a circle, for example, to test whether the distribution in longitude of something agrees with some theory, or whether two somethings have different distributions in longitude. (See also [8].)

We will leave to you the coding of routines analogous to `ksone`, `kstwo`, and `probks`, above. (For $\lambda < 0.4$, don't try to do the sum 14.3.14. Its value is 1, to 7 figures, but the series can require many terms to converge, and loses accuracy to roundoff.)

Two final cautionary notes: First, we should mention that all varieties of K–S test lack the ability to discriminate some kinds of distributions. A simple example is a probability distribution with a narrow "notch" within which the probability falls to zero. Such a distribution is of course ruled out by the existence of even one data point within the notch,

but, because of its cumulative nature, a K–S test would require many data points in the notch before signaling a discrepancy.

Second, we should note that, if you estimate any parameters from a data set (e.g., a mean and variance), then the distribution of the K–S statistic D for a cumulative distribution function $P(x)$ that *uses the estimated parameters* is no longer given by equation (14.3.9). In general, you will have to determine the new distribution yourself, e.g., by Monte Carlo methods.

CITED REFERENCES AND FURTHER READING:

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapters IX(C) and IX(E).

Stephens, M.A. 1970, *Journal of the Royal Statistical Society*, ser. B, vol. 32, pp. 115–122. [1]

Anderson, T.W., and Darling, D.A. 1952, *Annals of Mathematical Statistics*, vol. 23, pp. 193–212. [2]

Darling, D.A. 1957, *Annals of Mathematical Statistics*, vol. 28, pp. 823–838. [3]

Michael, J.R. 1983, *Biometrika*, vol. 70, no. 1, pp. 11–17. [4]

Noé, M. 1972, *Annals of Mathematical Statistics*, vol. 43, pp. 58–64. [5]

Kuiper, N.H. 1962, *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, ser. A., vol. 63, pp. 38–47. [6]

Stephens, M.A. 1965, *Biometrika*, vol. 52, pp. 309–321. [7]

Fisher, N.I., Lewis, T., and Embleton, B.J.J. 1987, *Statistical Analysis of Spherical Data* (New York: Cambridge University Press). [8]

14.4 Contingency Table Analysis of Two Distributions

In this section, and the next two sections, we deal with *measures of association* for two distributions. The situation is this: Each data point has two or more different quantities associated with it, and we want to know whether knowledge of one quantity gives us any demonstrable advantage in predicting the value of another quantity. In many cases, one variable will be an "independent" or "control" variable, and another will be a "dependent" or "measured" variable. Then, we want to know if the latter variable *is* in fact dependent on or *associated* with the former variable. If it is, we want to have some quantitative measure of the strength of the association. One often hears this loosely stated as the question of whether two variables are *correlated* or *uncorrelated*, but we will reserve those terms for a particular kind of association (linear, or at least monotonic), as discussed in §14.5 and §14.6.

Notice that, as in previous sections, the different concepts of significance and strength appear: The association between two distributions may be very significant even if that association is weak — if the quantity of data is large enough.

It is useful to distinguish among some different kinds of variables, with different categories forming a loose hierarchy.

- A variable is called *nominal* if its values are the members of some unordered set. For example, "state of residence" is a nominal variable that (in the U.S.) takes on one of 50 values; in astrophysics, "type of galaxy" is a nominal variable with the three values "spiral," "elliptical," and "irregular."

but, because of its cumulative nature, a K–S test would require many data points in the notch before signaling a discrepancy.

Second, we should note that, if you estimate any parameters from a data set (e.g., a mean and variance), then the distribution of the K–S statistic D for a cumulative distribution function $P(x)$ that *uses the estimated parameters* is no longer given by equation (14.3.9). In general, you will have to determine the new distribution yourself, e.g., by Monte Carlo methods.

CITED REFERENCES AND FURTHER READING:

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapters IX(C) and IX(E).

Stephens, M.A. 1970, *Journal of the Royal Statistical Society*, ser. B, vol. 32, pp. 115–122. [1]

Anderson, T.W., and Darling, D.A. 1952, *Annals of Mathematical Statistics*, vol. 23, pp. 193–212. [2]

Darling, D.A. 1957, *Annals of Mathematical Statistics*, vol. 28, pp. 823–838. [3]

Michael, J.R. 1983, *Biometrika*, vol. 70, no. 1, pp. 11–17. [4]

Noé, M. 1972, *Annals of Mathematical Statistics*, vol. 43, pp. 58–64. [5]

Kuiper, N.H. 1962, *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, ser. A., vol. 63, pp. 38–47. [6]

Stephens, M.A. 1965, *Biometrika*, vol. 52, pp. 309–321. [7]

Fisher, N.I., Lewis, T., and Embleton, B.J.J. 1987, *Statistical Analysis of Spherical Data* (New York: Cambridge University Press). [8]

14.4 Contingency Table Analysis of Two Distributions

In this section, and the next two sections, we deal with *measures of association* for two distributions. The situation is this: Each data point has two or more different quantities associated with it, and we want to know whether knowledge of one quantity gives us any demonstrable advantage in predicting the value of another quantity. In many cases, one variable will be an "independent" or "control" variable, and another will be a "dependent" or "measured" variable. Then, we want to know if the latter variable *is* in fact dependent on or *associated* with the former variable. If it is, we want to have some quantitative measure of the strength of the association. One often hears this loosely stated as the question of whether two variables are *correlated* or *uncorrelated*, but we will reserve those terms for a particular kind of association (linear, or at least monotonic), as discussed in §14.5 and §14.6.

Notice that, as in previous sections, the different concepts of significance and strength appear: The association between two distributions may be very significant even if that association is weak — if the quantity of data is large enough.

It is useful to distinguish among some different kinds of variables, with different categories forming a loose hierarchy.

- A variable is called *nominal* if its values are the members of some unordered set. For example, "state of residence" is a nominal variable that (in the U.S.) takes on one of 50 values; in astrophysics, "type of galaxy" is a nominal variable with the three values "spiral," "elliptical," and "irregular."

- A variable is termed *ordinal* if its values are the members of a discrete, but ordered, set. Examples are: grade in school, planetary order from the Sun (Mercury = 1, Venus = 2, ...), number of offspring. There need not be any concept of "equal metric distance" between the values of an ordinal variable, only that they be intrinsically ordered.
- We will call a variable *continuous* if its values are real numbers, as are times, distances, temperatures, etc. (Social scientists sometimes distinguish between *interval* and *ratio* continuous variables, but we do not find that distinction very compelling.)

A continuous variable can always be made into an ordinal one by binning it into ranges. If we choose to ignore the ordering of the bins, then we can turn it into a nominal variable. Nominal variables constitute the lowest type of the hierarchy, and therefore the most general. For example, a set of *several* continuous or ordinal variables can be turned, if crudely, into a single nominal variable, by coarsely binning each variable and then taking each distinct combination of bin assignments as a single nominal value. When multidimensional data are sparse, this is often the only sensible way to proceed.

The remainder of this section will deal with measures of association between *nominal* variables. For any pair of nominal variables, the data can be displayed as a *contingency table*, a table whose rows are labeled by the values of one nominal variable, whose columns are labeled by the values of the other nominal variable, and whose entries are nonnegative integers giving the number of observed events for each combination of row and column (see Figure 14.4.1). The analysis of association between nominal variables is thus called *contingency table analysis* or *crosstabulation analysis.*

We will introduce two different approaches. The first approach, based on the chi-square statistic, does a good job of characterizing the significance of association, but is only so-so as a measure of the strength (principally because its numerical values have no very direct interpretations). The second approach, based on the information-theoretic concept of *entropy*, says nothing at all about the significance of association (use chi-square for that!), but is capable of very elegantly characterizing the strength of an association already known to be significant.

Measures of Association Based on Chi-Square

Some notation first: Let N_{ij} denote the number of events that occur with the first variable x taking on its ith value, and the second variable y taking on its jth value. Let N denote the total number of events, the sum of all the N_{ij}'s. Let $N_{i\cdot}$ denote the number of events for which the first variable x takes on its ith value regardless of the value of y; $N_{\cdot j}$ is the number of events with the jth value of y regardless of x. So we have

$$N_{i\cdot} = \sum_j N_{ij} \qquad N_{\cdot j} = \sum_i N_{ij}$$
$$N = \sum_i N_{i\cdot} = \sum_j N_{\cdot j} \tag{14.4.1}$$

	1. red	2. green	. . .	
1. male	# of red males N_{11}	# of green males N_{12}	. . .	# of males $N_{1\cdot}$
2. female	# of red females N_{21}	# of green females N_{22}	. . .	# of females $N_{2\cdot}$
⋮	⋮	⋮	. . .	⋮
	# of red $N_{\cdot 1}$	# of green $N_{\cdot 2}$	. . .	total # N

Figure 14.4.1. Example of a contingency table for two nominal variables, here sex and color. The row and column marginals (totals) are shown. The variables are "nominal," i.e., the order in which their values are listed is arbitrary and does not affect the result of the contingency table analysis. If the ordering of values has some intrinsic meaning, then the variables are "ordinal" or "continuous," and correlation techniques (§14.5-§14.6) can be utilized.

$N_{\cdot j}$ and $N_{i\cdot}$ are sometimes called the *row and column totals* or *marginals*, but we will use these terms cautiously since we can never keep straight which are the rows and which are the columns!

The null hypothesis is that the two variables x and y have no association. In this case, the probability of a particular value of x given a particular value of y should be the same as the probability of that value of x regardless of y. Therefore, in the null hypothesis, the expected number for any N_{ij}, which we will denote n_{ij}, can be calculated from only the row and column totals,

$$\frac{n_{ij}}{N_{\cdot j}} = \frac{N_{i\cdot}}{N} \qquad \text{which implies} \qquad n_{ij} = \frac{N_{i\cdot} N_{\cdot j}}{N} \tag{14.4.2}$$

Notice that if a column or row total is zero, then the expected number for all the entries in that column or row is also zero; in that case, the never-occurring bin of x or y should simply be removed from the analysis.

The chi-square statistic is now given by equation (14.3.1), which, in the present case, is summed over all entries in the table,

$$\chi^2 = \sum_{i,j} \frac{(N_{ij} - n_{ij})^2}{n_{ij}} \tag{14.4.3}$$

The number of degrees of freedom is equal to the number of entries in the table (product of its row size and column size) minus the number of constraints that have arisen from our use of the data themselves to determine the n_{ij}. Each row total and column total is a constraint, except that this overcounts by one, since the total of the

column totals and the total of the row totals both equal N, the total number of data points. Therefore, if the table is of size I by J, the number of degrees of freedom is $IJ - I - J + 1$. Equation (14.4.3), along with the chi-square probability function (§6.2), now give the significance of an association between the variables x and y.

Suppose there is a significant association. How do we quantify its strength, so that (e.g.) we can compare the strength of one association with another? The idea here is to find some reparametrization of χ^2 which maps it into some convenient interval, like 0 to 1, where the result is not dependent on the quantity of data that we happen to sample, but rather depends only on the underlying population from which the data were drawn. There are several different ways of doing this. Two of the more common are called *Cramer's V* and the *contingency coefficient C*.

The formula for Cramer's V is

$$V = \sqrt{\frac{\chi^2}{N \min(I-1, J-1)}} \tag{14.4.4}$$

where I and J are again the numbers of rows and columns, and N is the total number of events. Cramer's V has the pleasant property that it lies between zero and one inclusive, equals zero when there is no association, and equals one only when the association is perfect: All the events in any row lie in one unique column, and vice versa. (In chess parlance, no two rooks, placed on a nonzero table entry, can capture each other.)

In the case of $I = J = 2$, Cramer's V is also referred to as the *phi* statistic.

The contingency coefficient C is defined as

$$C = \sqrt{\frac{\chi^2}{\chi^2 + N}} \tag{14.4.5}$$

It also lies between zero and one, but (as is apparent from the formula) it can never achieve the upper limit. While it can be used to compare the strength of association of two tables with the same I and J, its upper limit depends on I and J. Therefore it can never be used to compare tables of different sizes.

The trouble with both Cramer's V and the contingency coefficient C is that, when they take on values in between their extremes, there is no very direct interpretation of what that value means. For example, you are in Las Vegas, and a friend tells you that there is a small, but significant, association between the color of a croupier's eyes and the occurrence of red and black on his roulette wheel. Cramer's V is about 0.028, your friend tells you. You know what the usual odds against you are (because of the green zero and double zero on the wheel). Is this association sufficient for you to make money? Don't ask us!

```
      SUBROUTINE cntab1(nn,ni,nj,chisq,df,prob,cramrv,ccc)
      INTEGER ni,nj,nn(ni,nj),MAXI,MAXJ
      REAL ccc,chisq,cramrv,df,prob,TINY
      PARAMETER (MAXI=100,MAXJ=100,TINY=1.e-30)     Maximum table size, and a small num-
C     USES gammq                                      ber.
```

Given a two-dimensional contingency table in the form of an integer array `nn(1:ni,1:nj)`, this routine returns the chi-square `chisq`, the number of degrees of freedom `df`, the significance level `prob` (small values indicating a significant association), and two measures of association, Cramer's V (`cramrv`) and the contingency coefficient C (`ccc`).

```
      INTEGER i,j,nni,nnj
```

```
REAL expctd,sum,sumi(MAXI),sumj(MAXJ),gammq
sum=0                                          Will be total number of events.
nni=ni                                         Number of rows
nnj=nj                                         and columns.
do 12 i=1,ni                                   Get the row totals.
    sumi(i)=0.
    do 11 j=1,nj
        sumi(i)=sumi(i)+nn(i,j)
        sum=sum+nn(i,j)
    enddo 11
    if(sumi(i).eq.0.)nni=nni-1                 Eliminate any zero rows by reducing the
enddo 12                                           number.
do 14 j=1,nj                                   Get the column totals.
    sumj(j)=0.
    do 13 i=1,ni
        sumj(j)=sumj(j)+nn(i,j)
    enddo 13
    if(sumj(j).eq.0.)nnj=nnj-1                 Eliminate any zero columns.
enddo 14
df=nni*nnj-nni-nnj+1                           Corrected number of degrees of freedom.
chisq=0.
do 16 i=1,ni                                   Do the chi-square sum.
    do 15 j=1,nj
        expctd=sumj(j)*sumi(i)/sum
        chisq=chisq+(nn(i,j)-expctd)**2/(expctd+TINY)   Here TINY guarantees that
    enddo 15                                   any eliminated row or column will not
enddo 16                                       contribute to the sum.
prob=gammq(0.5*df,0.5*chisq)                   Chi-square probability function.
cramrv=sqrt(chisq/(sum*min(nni-1,nnj-1)))
ccc=sqrt(chisq/(chisq+sum))
return
END
```

Measures of Association Based on Entropy

Consider the game of "twenty questions," where by repeated yes/no questions you try to eliminate all except one correct possibility for an unknown object. Better yet, consider a generalization of the game, where you are allowed to ask multiple choice questions as well as binary (yes/no) ones. The categories in your multiple choice questions are supposed to be mutually exclusive and exhaustive (as are "yes" and "no").

The value to you of an answer increases with the number of possibilities that it eliminates. More specifically, an answer that eliminates all except a fraction p of the remaining possibilities can be assigned a value $-\ln p$ (a positive number, since $p < 1$). The purpose of the logarithm is to make the value additive, since (e.g.) one question that eliminates all but 1/6 of the possibilities is considered as good as two questions that, in sequence, reduce the number by factors 1/2 and 1/3.

So that is the value of an answer; but what is the value of a question? If there are I possible answers to the question $(i = 1, \ldots, I)$ and the fraction of possibilities consistent with the ith answer is p_i (with the sum of the p_i's equal to one), then the value of the question is the expectation value of the value of the answer, denoted H,

$$H = -\sum_{i=1}^{I} p_i \ln p_i \tag{14.4.6}$$

In evaluating (14.4.6), note that

$$\lim_{p\to 0} p\ln p = 0 \tag{14.4.7}$$

The value H lies between 0 and $\ln I$. It is zero only when one of the p_i's is one, all the others zero: In this case, the question is valueless, since its answer is preordained. H takes on its maximum value when all the p_i's are equal, in which case the question is sure to eliminate all but a fraction $1/I$ of the remaining possibilities.

The value H is conventionally termed the *entropy* of the distribution given by the p_i's, a terminology borrowed from statistical physics.

So far we have said nothing about the association of two variables; but suppose we are deciding what question to ask next in the game and have to choose between two candidates, or possibly want to ask both in one order or another. Suppose that one question, x, has I possible answers, labeled by i, and that the other question, y, as J possible answers, labeled by j. Then the possible outcomes of asking both questions form a contingency table whose entries N_{ij}, when normalized by dividing by the total number of remaining possibilities N, give all the information about the p's. In particular, we can make contact with the notation (14.4.1) by identifying

$$\begin{aligned} p_{ij} &= \frac{N_{ij}}{N} \\ p_{i\cdot} &= \frac{N_{i\cdot}}{N} \qquad \text{(outcomes of question } x \text{ alone)} \\ p_{\cdot j} &= \frac{N_{\cdot j}}{N} \qquad \text{(outcomes of question } y \text{ alone)} \end{aligned} \tag{14.4.8}$$

The entropies of the questions x and y are, respectively,

$$H(x) = -\sum_i p_{i\cdot}\ln p_{i\cdot} \qquad H(y) = -\sum_j p_{\cdot j}\ln p_{\cdot j} \tag{14.4.9}$$

The entropy of the two questions together is

$$H(x,y) = -\sum_{i,j} p_{ij}\ln p_{ij} \tag{14.4.10}$$

Now what is the entropy of the question y *given* x (that is, if x is asked first)? It is the expectation value over the answers to x of the entropy of the restricted y distribution that lies in a single column of the contingency table (corresponding to the x answer):

$$H(y|x) = -\sum_i p_{i\cdot}\sum_j \frac{p_{ij}}{p_{i\cdot}}\ln\frac{p_{ij}}{p_{i\cdot}} = -\sum_{i,j} p_{ij}\ln\frac{p_{ij}}{p_{i\cdot}} \tag{14.4.11}$$

Correspondingly, the entropy of x given y is

$$H(x|y) = -\sum_j p_{\cdot j}\sum_i \frac{p_{ij}}{p_{\cdot j}}\ln\frac{p_{ij}}{p_{\cdot j}} = -\sum_{i,j} p_{ij}\ln\frac{p_{ij}}{p_{\cdot j}} \tag{14.4.12}$$

We can readily prove that the entropy of y given x is never more than the entropy of y alone, i.e., that asking x first can only reduce the usefulness of asking y (in which case the two variables are *associated*!):

$$\begin{aligned} H(y|x) - H(y) &= -\sum_{i,j} p_{ij} \ln \frac{p_{ij}/p_{i\cdot}}{p_{\cdot j}} \\ &= \sum_{i,j} p_{ij} \ln \frac{p_{\cdot j} p_{i\cdot}}{p_{ij}} \\ &\le \sum_{i,j} p_{ij} \left(\frac{p_{\cdot j} p_{i\cdot}}{p_{ij}} - 1 \right) \\ &= \sum_{i,j} p_{i\cdot} p_{\cdot j} - \sum_{i,j} p_{ij} \\ &= 1 - 1 = 0 \end{aligned} \tag{14.4.13}$$

where the inequality follows from the fact

$$\ln w \le w - 1 \tag{14.4.14}$$

We now have everything we need to define a measure of the "dependency" of y on x, that is to say a measure of association. This measure is sometimes called the *uncertainty coefficient* of y. We will denote it as $U(y|x)$,

$$U(y|x) \equiv \frac{H(y) - H(y|x)}{H(y)} \tag{14.4.15}$$

This measure lies between zero and one, with the value 0 indicating that x and y have no association, the value 1 indicating that knowledge of x completely predicts y. For in-between values, $U(y|x)$ gives the fraction of y's entropy $H(y)$ that is lost if x is already known (i.e., that is redundant with the information in x). In our game of "twenty questions," $U(y|x)$ is the fractional loss in the utility of question y if question x is to be asked first.

If we wish to view x as the dependent variable, y as the independent one, then interchanging x and y we can of course define the dependency of x on y,

$$U(x|y) \equiv \frac{H(x) - H(x|y)}{H(x)} \tag{14.4.16}$$

If we want to treat x and y symmetrically, then the useful combination turns out to be

$$U(x,y) \equiv 2 \left[\frac{H(y) + H(x) - H(x,y)}{H(x) + H(y)} \right] \tag{14.4.17}$$

If the two variables are completely independent, then $H(x,y) = H(x) + H(y)$, so (14.4.17) vanishes. If the two variables are completely dependent, then $H(x) = H(y) = H(x,y)$, so (14.4.16) equals unity. In fact, you can use the identities (easily proved from equations 14.4.9–14.4.12)

$$H(x,y) = H(x) + H(y|x) = H(y) + H(x|y) \tag{14.4.18}$$

to show that

$$U(x,y) = \frac{H(x)U(x|y) + H(y)U(y|x)}{H(x) + H(y)} \tag{14.4.19}$$

i.e., that the symmetrical measure is just a weighted average of the two asymmetrical measures (14.4.15) and (14.4.16), weighted by the entropy of each variable separately.

Here is a program for computing all the quantities discussed, $H(x)$, $H(y)$, $H(x|y)$, $H(y|x)$, $H(x,y)$, $U(x|y)$, $U(y|x)$, and $U(x,y)$:

```
SUBROUTINE cntab2(nn,ni,nj,h,hx,hy,hygx,hxgy,uygx,uxgy,uxy)
INTEGER ni,nj,nn(ni,nj),MAXI,MAXJ
REAL h,hx,hxgy,hy,hygx,uxgy,uxy,uygx,TINY
PARAMETER (MAXI=100,MAXJ=100,TINY=1.e-30)
```

Given a two-dimensional contingency table in the form of an integer array `nn(i,j)`, where `i` labels the x variable and ranges from 1 to `ni`, `j` labels the y variable and ranges from 1 to `nj`, this routine returns the entropy `h` of the whole table, the entropy `hx` of the x distribution, the entropy `hy` of the y distribution, the entropy `hygx` of y given x, the entropy `hxgy` of x given y, the dependency `uygx` of y on x (eq. 14.4.15), the dependency `uxgy` of x on y (eq. 14.4.16), and the symmetrical dependency `uxy` (eq. 14.4.17).

Parameters: `MAXI` and `MAXJ` define the maximum size of table; `TINY` is a small number.

```
INTEGER i,j
REAL p,sum,sumi(MAXI),sumj(MAXJ)
sum=0
do 12 i=1,ni                                  Get the row totals.
    sumi(i)=0.0
    do 11 j=1,nj
        sumi(i)=sumi(i)+nn(i,j)
        sum=sum+nn(i,j)
    enddo 11
enddo 12
do 14 j=1,nj                                  Get the column totals.
    sumj(j)=0.
    do 13 i=1,ni
        sumj(j)=sumj(j)+nn(i,j)
    enddo 13
enddo 14
hx=0.                                         Entropy of the x distribution,
do 15 i=1,ni
    if(sumi(i).ne.0.)then
        p=sumi(i)/sum
        hx=hx-p*log(p)
    endif
enddo 15
hy=0.                                         and of the y distribution.
do 16 j=1,nj
    if(sumj(j).ne.0.)then
        p=sumj(j)/sum
        hy=hy-p*log(p)
    endif
enddo 16
h=0.
do 18 i=1,ni                                  Total entropy: loop over both x
    do 17 j=1,nj                              and y.
        if(nn(i,j).ne.0)then
            p=nn(i,j)/sum
            h=h-p*log(p)
        endif
    enddo 17
enddo 18
hygx=h-hx                                     Uses equation (14.4.18),
hxgy=h-hy                                     as does this.
```

```
uygx=(hy-hygx)/(hy+TINY)                Equation (14.4.15).
uxgy=(hx-hxgy)/(hx+TINY)                Equation (14.4.16).
uxy=2.*(hx+hy-h)/(hx+hy+TINY)           Equation (14.4.17).
return
END
```

CITED REFERENCES AND FURTHER READING:

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Fano, R.M. 1961, *Transmission of Information* (New York: Wiley and MIT Press), Chapter 2.

14.5 Linear Correlation

We next turn to measures of association between variables that are ordinal or continuous, rather than nominal. Most widely used is the *linear correlation coefficient*. For pairs of quantities (x_i, y_i), $i = 1, \ldots, N$, the linear correlation coefficient r (also called the product-moment correlation coefficient, or *Pearson's r*) is given by the formula

$$r = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_i (x_i - \overline{x})^2} \sqrt{\sum_i (y_i - \overline{y})^2}} \tag{14.5.1}$$

where, as usual, $\overline{x}$ is the mean of the x_i's, $\overline{y}$ is the mean of the y_i's.

The value of r lies between -1 and 1, inclusive. It takes on a value of 1, termed "complete positive correlation," when the data points lie on a perfect straight line with positive slope, with x and y increasing together. The value 1 holds independent of the magnitude of the slope. If the data points lie on a perfect straight line with negative slope, y decreasing as x increases, then r has the value -1; this is called "complete negative correlation." A value of r near zero indicates that the variables x and y are *uncorrelated*.

When a correlation is known to be significant, r is one conventional way of summarizing its strength. In fact, the value of r can be translated into a statement about what residuals (root mean square deviations) are to be expected if the data are fitted to a straight line by the least-squares method (see §15.2, especially equations 15.2.13 – 15.2.14). Unfortunately, r is a rather poor statistic for deciding *whether* an observed correlation is statistically significant, and/or whether one observed correlation is significantly stronger than another. The reason is that r is ignorant of the individual distributions of x and y, so there is no universal way to compute its distribution in the case of the null hypothesis.

About the only general statement that can be made is this: If the null hypothesis is that x and y are uncorrelated, and if the distributions for x and y each have enough convergent moments ("tails" die off sufficiently rapidly), and if N is large

```
uygx=(hy-hygx)/(hy+TINY)                Equation (14.4.15).
uxgy=(hx-hxgy)/(hx+TINY)                Equation (14.4.16).
uxy=2.*(hx+hy-h)/(hx+hy+TINY)           Equation (14.4.17).
return
END
```

CITED REFERENCES AND FURTHER READING:

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

Fano, R.M. 1961, *Transmission of Information* (New York: Wiley and MIT Press), Chapter 2.

14.5 Linear Correlation

We next turn to measures of association between variables that are ordinal or continuous, rather than nominal. Most widely used is the *linear correlation coefficient*. For pairs of quantities (x_i, y_i), $i = 1, \ldots, N$, the linear correlation coefficient r (also called the product-moment correlation coefficient, or *Pearson's r*) is given by the formula

$$r = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_i (x_i - \overline{x})^2} \sqrt{\sum_i (y_i - \overline{y})^2}} \tag{14.5.1}$$

where, as usual, $\overline{x}$ is the mean of the x_i's, $\overline{y}$ is the mean of the y_i's.

The value of r lies between -1 and 1, inclusive. It takes on a value of 1, termed "complete positive correlation," when the data points lie on a perfect straight line with positive slope, with x and y increasing together. The value 1 holds independent of the magnitude of the slope. If the data points lie on a perfect straight line with negative slope, y decreasing as x increases, then r has the value -1; this is called "complete negative correlation." A value of r near zero indicates that the variables x and y are *uncorrelated*.

When a correlation is known to be significant, r is one conventional way of summarizing its strength. In fact, the value of r can be translated into a statement about what residuals (root mean square deviations) are to be expected if the data are fitted to a straight line by the least-squares method (see §15.2, especially equations 15.2.13 – 15.2.14). Unfortunately, r is a rather poor statistic for deciding *whether* an observed correlation is statistically significant, and/or whether one observed correlation is significantly stronger than another. The reason is that r is ignorant of the individual distributions of x and y, so there is no universal way to compute its distribution in the case of the null hypothesis.

About the only general statement that can be made is this: If the null hypothesis is that x and y are uncorrelated, and if the distributions for x and y each have enough convergent moments ("tails" die off sufficiently rapidly), and if N is large

(typically > 500), then r is distributed approximately normally, with a mean of zero and a standard deviation of $1/\sqrt{N}$. In that case, the (double-sided) significance of the correlation, that is, the probability that $|r|$ should be larger than its observed value in the null hypothesis, is

$$\text{erfc}\left(\frac{|r|\sqrt{N}}{\sqrt{2}}\right) \tag{14.5.2}$$

where $\text{erfc}(x)$ is the complementary error function, equation (6.2.8), computed by the routines `erfc` or `erfcc` of §6.2. A small value of (14.5.2) indicates that the two distributions are significantly correlated. (See expression 14.5.9 below for a more accurate test.)

Most statistics books try to go beyond (14.5.2) and give additional statistical tests that can be made using r. In almost all cases, however, these tests are valid only for a very special class of hypotheses, namely that the distributions of x and y jointly form a *binormal* or *two-dimensional Gaussian* distribution around their mean values, with joint probability density

$$p(x,y)\,dxdy = \text{const.} \times \exp\left[-\frac{1}{2}(a_{11}x^2 - 2a_{12}xy + a_{22}y^2)\right]\,dxdy \tag{14.5.3}$$

where a_{11}, a_{12}, and a_{22} are arbitrary constants. For this distribution r has the value

$$r = -\frac{a_{12}}{\sqrt{a_{11}a_{22}}} \tag{14.5.4}$$

There are occasions when (14.5.3) may be known to be a good model of the data. There may be other occasions when we are willing to take (14.5.3) as at least a rough and ready guess, since many two-dimensional distributions do resemble a binormal distribution, at least not too far out on their tails. In either situation, we can use (14.5.3) to go beyond (14.5.2) in any of several directions:

First, we can allow for the possibility that the number N of data points is not large. Here, it turns out that the statistic

$$t = r\sqrt{\frac{N-2}{1-r^2}} \tag{14.5.5}$$

is distributed in the null case (of no correlation) like Student's t-distribution with $\nu = N - 2$ degrees of freedom, whose two-sided significance level is given by $1 - A(t|\nu)$ (equation 6.4.7). As N becomes large, this significance and (14.5.2) become asymptotically the same, so that one never does worse by using (14.5.5), even if the binormal assumption is not well substantiated.

Second, when N is only moderately large (≥ 10), we can compare whether the difference of two significantly nonzero r's, e.g., from different experiments, is itself significant. In other words, we can quantify whether a change in some control variable significantly alters an existing correlation between two other variables. This is done by using *Fisher's z-transformation* to associate each measured r with a corresponding z,

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right) \tag{14.5.6}$$

Then, each z is approximately normally distributed with a mean value

$$\overline{z} = \frac{1}{2}\left[\ln\left(\frac{1+r_{\rm true}}{1-r_{\rm true}}\right) + \frac{r_{\rm true}}{N-1}\right] \tag{14.5.7}$$

where $r_{\rm true}$ is the actual or population value of the correlation coefficient, and with a standard deviation

$$\sigma(z) \approx \frac{1}{\sqrt{N-3}} \tag{14.5.8}$$

Equations (14.5.7) and (14.5.8), when they are valid, give several useful statistical tests. For example, the significance level at which a measured value of r differs from some hypothesized value $r_{\rm true}$ is given by

$$\text{erfc}\left(\frac{|z-\overline{z}|\sqrt{N-3}}{\sqrt{2}}\right) \tag{14.5.9}$$

where z and $\overline{z}$ are given by (14.5.6) and (14.5.7), with small values of (14.5.9) indicating a significant difference. (Setting $\overline{z}=0$ makes expression 14.5.9 a more accurate replacement for expression 14.5.2 above.) Similarly, the significance of a difference between two measured correlation coefficients r_1 and r_2 is

$$\text{erfc}\left(\frac{|z_1-z_2|}{\sqrt{2}\sqrt{\frac{1}{N_1-3}+\frac{1}{N_2-3}}}\right) \tag{14.5.10}$$

where z_1 and z_2 are obtained from r_1 and r_2 using (14.5.6), and where N_1 and N_2 are, respectively, the number of data points in the measurement of r_1 and r_2.

All of the significances above are two-sided. If you wish to disprove the null hypothesis in favor of a one-sided hypothesis, such as that $r_1 > r_2$ (where the sense of the inequality was decided *a priori*), then (i) if your measured r_1 and r_2 have the *wrong* sense, you have failed to demonstrate your one-sided hypothesis, but (ii) if they have the right ordering, you can multiply the significances given above by 0.5, which makes them more significant.

But keep in mind: These interpretations of the r statistic can be completely meaningless if the joint probability distribution of your variables x and y is too different from a binormal distribution.

```
      SUBROUTINE pearsn(x,y,n,r,prob,z)
      INTEGER n
      REAL prob,r,z,x(n),y(n),TINY
      PARAMETER (TINY=1.e-20)                    Will regularize the unusual case of com-
C     USES betai                                   plete correlation.
```

Given two arrays `x(1:n)` and `y(1:n)`, this routine computes their correlation coefficient r (returned as `r`), the significance level at which the null hypothesis of zero correlation is disproved (`prob` whose small value indicates a significant correlation), and Fisher's z (returned as `z`), whose value can be used in further statistical tests as described above.

```
      INTEGER j
      REAL ax,ay,df,sxx,sxy,syy,t,xt,yt,betai
      ax=0.
      ay=0.
      do 11 j=1,n                                Find the means.
```

```
        ax=ax+x(j)
        ay=ay+y(j)
    enddo 11
    ax=ax/n
    ay=ay/n
    sxx=0.
    syy=0.
    sxy=0.
    do 12 j=1,n                                  Compute the correlation coefficient.
        xt=x(j)-ax
        yt=y(j)-ay
        sxx=sxx+xt**2
        syy=syy+yt**2
        sxy=sxy+xt*yt
    enddo 12
    r=sxy/(sqrt(sxx*syy)+TINY)
    z=0.5*log(((1.+r)+TINY)/((1.-r)+TINY))       Fisher's z transformation.
    df=n-2
    t=r*sqrt(df/(((1.-r)+TINY)*((1.+r)+TINY)))   Equation (14.5.5).
    prob=betai(0.5*df,0.5,df/(df+t**2))          Student's t probability.
C   prob=erfcc(abs(z*sqrt(n-1.))/1.4142136)      For large n, this easier computation of
    return                                           prob, using the short routine erfcc,
    END                                              would give approximately the same
                                                     value.
```

CITED REFERENCES AND FURTHER READING:

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

Hoel, P.G. 1971, *Introduction to Mathematical Statistics*, 4th ed. (New York: Wiley), Chapter 7.

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapters IX(A) and IX(B).

Korn, G.A., and Korn, T.M. 1968, *Mathematical Handbook for Scientists and Engineers*, 2nd ed. (New York: McGraw-Hill), §19.7.

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

14.6 Nonparametric or Rank Correlation

It is precisely the uncertainty in interpreting the significance of the linear correlation coefficient r that leads us to the important concepts of *nonparametric* or *rank correlation*. As before, we are given N pairs of measurements (x_i, y_i). Before, difficulties arose because we did not necessarily know the probability distribution function from which the x_i's or y_i's were drawn.

The key concept of nonparametric correlation is this: If we replace the value of each x_i by the value of its *rank* among all the other x_i's in the sample, that is, $1, 2, 3, \ldots, N$, then the resulting list of numbers will be drawn from a perfectly known distribution function, namely uniformly from the integers between 1 and N, inclusive. Better than uniformly, in fact, since if the x_i's are all distinct, then each integer will occur precisely once. If some of the x_i's have identical values, it is conventional to assign to all these "ties" the mean of the ranks that they would have had if their values had been slightly different. This *midrank* will sometimes be an

```
        ax=ax+x(j)
        ay=ay+y(j)
    enddo 11
    ax=ax/n
    ay=ay/n
    sxx=0.
    syy=0.
    sxy=0.
    do 12 j=1,n                                  Compute the correlation coefficient.
        xt=x(j)-ax
        yt=y(j)-ay
        sxx=sxx+xt**2
        syy=syy+yt**2
        sxy=sxy+xt*yt
    enddo 12
    r=sxy/(sqrt(sxx*syy)+TINY)
    z=0.5*log(((1.+r)+TINY)/((1.-r)+TINY))       Fisher's z transformation.
    df=n-2
    t=r*sqrt(df/(((1.-r)+TINY)*((1.+r)+TINY)))   Equation (14.5.5).
    prob=betai(0.5*df,0.5,df/(df+t**2))          Student's t probability.
C   prob=erfcc(abs(z*sqrt(n-1.))/1.4142136)      For large n, this easier computation of
    return                                           prob, using the short routine erfcc,
    END                                              would give approximately the same
                                                     value.
```

CITED REFERENCES AND FURTHER READING:

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

Hoel, P.G. 1971, *Introduction to Mathematical Statistics*, 4th ed. (New York: Wiley), Chapter 7.

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapters IX(A) and IX(B).

Korn, G.A., and Korn, T.M. 1968, *Mathematical Handbook for Scientists and Engineers*, 2nd ed. (New York: McGraw-Hill), §19.7.

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

14.6 Nonparametric or Rank Correlation

It is precisely the uncertainty in interpreting the significance of the linear correlation coefficient r that leads us to the important concepts of *nonparametric* or *rank correlation*. As before, we are given N pairs of measurements (x_i, y_i). Before, difficulties arose because we did not necessarily know the probability distribution function from which the x_i's or y_i's were drawn.

The key concept of nonparametric correlation is this: If we replace the value of each x_i by the value of its *rank* among all the other x_i's in the sample, that is, $1, 2, 3, \ldots, N$, then the resulting list of numbers will be drawn from a perfectly known distribution function, namely uniformly from the integers between 1 and N, inclusive. Better than uniformly, in fact, since if the x_i's are all distinct, then each integer will occur precisely once. If some of the x_i's have identical values, it is conventional to assign to all these "ties" the mean of the ranks that they would have had if their values had been slightly different. This *midrank* will sometimes be an

integer, sometimes a half-integer. In all cases the sum of all assigned ranks will be the same as the sum of the integers from 1 to N, namely $\frac{1}{2}N(N+1)$.

Of course we do exactly the same procedure for the y_i's, replacing each value by its rank among the other y_i's in the sample.

Now we are free to invent statistics for detecting correlation between uniform sets of integers between 1 and N, keeping in mind the possibility of ties in the ranks. There is, of course, some loss of information in replacing the original numbers by ranks. We could construct some rather artificial examples where a correlation could be detected parametrically (e.g., in the linear correlation coefficient r), but could not be detected nonparametrically. Such examples are very rare in real life, however, and the slight loss of information in ranking is a small price to pay for a very major advantage: When a correlation is demonstrated to be present nonparametrically, then it is really there! (That is, to a certainty level that depends on the significance chosen.) Nonparametric correlation is more robust than linear correlation, more resistant to unplanned defects in the data, in the same sort of sense that the median is more robust than the mean. For more on the concept of robustness, see §15.7.

As always in statistics, some particular choices of a statistic have already been invented for us and consecrated, if not beatified, by popular use. We will discuss two, the *Spearman rank-order correlation coefficient* (r_s), and *Kendall's tau* (τ).

Spearman Rank-Order Correlation Coefficient

Let R_i be the rank of x_i among the other x's, S_i be the rank of y_i among the other y's, ties being assigned the appropriate midrank as described above. Then the rank-order correlation coefficient is defined to be the linear correlation coefficient of the ranks, namely,

$$r_s = \frac{\sum_i (R_i - \overline{R})(S_i - \overline{S})}{\sqrt{\sum_i (R_i - \overline{R})^2} \sqrt{\sum_i (S_i - \overline{S})^2}} \tag{14.6.1}$$

The significance of a nonzero value of r_s is tested by computing

$$t = r_s \sqrt{\frac{N-2}{1-r_s^2}} \tag{14.6.2}$$

which is distributed approximately as Student's distribution with $N-2$ degrees of freedom. A key point is that this approximation does not depend on the original distribution of the x's and y's; it is always the same approximation, and always pretty good.

It turns out that r_s is closely related to another conventional measure of nonparametric correlation, the so-called *sum squared difference of ranks*, defined as

$$D = \sum_{i=1}^{N} (R_i - S_i)^2 \tag{14.6.3}$$

(This D is sometimes denoted D^{**}, where the asterisks are used to indicate that ties are treated by midranking.)

When there are no ties in the data, then the exact relation between D and r_s is

$$r_s = 1 - \frac{6D}{N^3 - N} \tag{14.6.4}$$

When there are ties, then the exact relation is slightly more complicated: Let f_k be the number of ties in the kth group of ties among the R_i's, and let g_m be the number of ties in the mth group of ties among the S_i's. Then it turns out that

$$r_s = \frac{1 - \frac{6}{N^3 - N}\left[D + \frac{1}{12}\sum_k (f_k^3 - f_k) + \frac{1}{12}\sum_m (g_m^3 - g_m)\right]}{\left[1 - \frac{\sum_k (f_k^3 - f_k)}{N^3 - N}\right]^{1/2}\left[1 - \frac{\sum_m (g_m^3 - g_m)}{N^3 - N}\right]^{1/2}} \tag{14.6.5}$$

holds exactly. Notice that if all the f_k's and all the g_m's are equal to one, meaning that there are no ties, then equation (14.6.5) reduces to equation (14.6.4).

In (14.6.2) we gave a t-statistic that tests the significance of a nonzero r_s. It is also possible to test the significance of D directly. The expectation value of D in the null hypothesis of uncorrelated data sets is

$$\overline{D} = \frac{1}{6}(N^3 - N) - \frac{1}{12}\sum_k (f_k^3 - f_k) - \frac{1}{12}\sum_m (g_m^3 - g_m) \tag{14.6.6}$$

its variance is

$$\begin{aligned}\mathrm{Var}(D) = {} & \frac{(N-1)N^2(N+1)^2}{36} \\ & \times \left[1 - \frac{\sum_k (f_k^3 - f_k)}{N^3 - N}\right]\left[1 - \frac{\sum_m (g_m^3 - g_m)}{N^3 - N}\right]\end{aligned} \tag{14.6.7}$$

and it is approximately normally distributed, so that the significance level is a complementary error function (cf. equation 14.5.2). Of course, (14.6.2) and (14.6.7) are not independent tests, but simply variants of the same test. In the program that follows, we return both the significance level obtained by using (14.6.2) and the significance level obtained by using (14.6.7); their discrepancy will give you an idea of how good the approximations are. You will also notice that we break off the task of assigning ranks (including tied midranks) into a separate routine, `crank`.

```
      SUBROUTINE spear(data1,data2,n,wksp1,wksp2,d,zd,probd,rs,probrs)
      INTEGER n
      REAL d,probd,probrs,rs,zd,data1(n),data2(n),wksp1(n),wksp2(n)
C     USES betai,crank,erfcc,sort2
```

Given two data arrays, `data1(1:n)` and `data2(1:n)`, each of length `n`, and given two workspaces of the same length, this routine returns their sum-squared difference of ranks as D, the number of standard deviations by which D deviates from its null-hypothesis expected value as `zd`, the two-sided significance level of this deviation as `probd`, Spearman's rank correlation r_s as `rs`, and the two-sided significance level of its deviation from zero as `probrs`. The workspaces can be identical to the data arrays, but in that case the data arrays are destroyed. The external routines `crank` (below) and `sort2` (§8.2) are used. A

small value of either probd or probrs indicates a significant correlation (rs positive) or anticorrelation (rs negative).

```
      INTEGER j
      REAL aved,df,en,en3n,fac,sf,sg,t,vard,betai,erfcc
      do 11 j=1,n
          wksp1(j)=data1(j)
          wksp2(j)=data2(j)
      enddo 11
      call sort2(n,wksp1,wksp2)
      call crank(n,wksp1,sf)
      call sort2(n,wksp2,wksp1)
      call crank(n,wksp2,sg)
      d=0.
      do 12 j=1,n
          d=d+(wksp1(j)-wksp2(j))**2
      enddo 12
      en=n
      en3n=en**3-en
      aved=en3n/6.-(sf+sg)/12.
      fac=(1.-sf/en3n)*(1.-sg/en3n)
      vard=((en-1.)*en**2*(en+1.)**2/36.)*fac
      zd=(d-aved)/sqrt(vard)
      probd=erfcc(abs(zd)/1.4142136)
      rs=(1.-(6./en3n)*(d+(sf+sg)/12.))/sqrt(fac)
      fac=(1.+rs)*(1.-rs)
      if(fac.gt.0.)then
          t=rs*sqrt((en-2.)/fac)
          df=en-2.
          probrs=betai(0.5*df,0.5,df/(df+t**2))
      else
          probrs=0.
      endif
      return
      END
```

Sort each of the data arrays, and convert the entries to ranks. The values sf and sg return the sums $\sum(f_k^3 - f_k)$ and $\sum(g_m^3 - g_m)$, respectively.

Sum the squared difference of ranks.

Expectation value of D,

and variance of D give

number of standard deviations,

and significance.

Rank correlation coefficient,

and its t value,

give its significance.

```
      SUBROUTINE crank(n,w,s)
      INTEGER n
      REAL s,w(n)
```

Given a sorted array w(1:n), replaces the elements by their rank, including midranking of ties, and returns as s the sum of $f^3 - f$, where f is the number of elements in each tie.

```
      INTEGER j,ji,jt
      REAL rank,t
      s=0.
      j=1                              The next rank to be assigned.
1     if(j.lt.n)then                   "do while" structure.
          if(w(j+1).ne.w(j))then       Not a tie.
              w(j)=j
              j=j+1
          else                         A tie:
              do 11 jt=j+1,n           How far does it go?
                  if(w(jt).ne.w(j))goto 2
              enddo 11
              jt=n+1                   If here, it goes all the way to the last element.
2             rank=0.5*(j+jt-1)        This is the mean rank of the tie,
              do 12 ji=j,jt-1          so enter it into all the tied entries,
                  w(ji)=rank
              enddo 12
              t=jt-j
              s=s+t**3-t               and update s.
              j=jt
          endif
      goto 1
```

```
endif
if(j.eq.n)w(n)=n                    If the last element was not tied, this is its rank.
return
END
```

Kendall's Tau

Kendall's τ is even more nonparametric than Spearman's r_s or D. Instead of using the numerical difference of ranks, it uses only the relative ordering of ranks: higher in rank, lower in rank, or the same in rank. But in that case we don't even have to rank the data! Ranks will be higher, lower, or the same if and only if the values are larger, smaller, or equal, respectively. On balance, we prefer r_s as being the more straightforward nonparametric test, but both statistics are in general use. In fact, τ and r_s are very strongly correlated and, in most applications, are effectively the same test.

To define τ, we start with the N data points (x_i, y_i). Now consider all $\frac{1}{2}N(N-1)$ *pairs* of data points, where a data point cannot be paired with itself, and where the points in either order count as one pair. We call a pair *concordant* if the relative ordering of the ranks of the two x's (or for that matter the two x's themselves) is the same as the relative ordering of the ranks of the two y's (or for that matter the two y's themselves). We call a pair *discordant* if the relative ordering of the ranks of the two x's is opposite from the relative ordering of the ranks of the two y's. If there is a tie in either the ranks of the two x's or the ranks of the two y's, then we don't call the pair either concordant or discordant. If the tie is in the x's, we will call the pair an "extra y pair." If the tie is in the y's, we will call the pair an "extra x pair." If the tie is in both the x's and the y's, we don't call the pair anything at all. Are you still with us?

Kendall's τ is now the following simple combination of these various counts:

$$\tau = \frac{\text{concordant} - \text{discordant}}{\sqrt{\text{concordant} + \text{discordant} + \text{extra-}y}\ \sqrt{\text{concordant} + \text{discordant} + \text{extra-}x}} \tag{14.6.8}$$

You can easily convince yourself that this must lie between 1 and -1, and that it takes on the extreme values only for complete rank agreement or complete rank reversal, respectively.

More important, Kendall has worked out, from the combinatorics, the approximate distribution of τ in the null hypothesis of no association between x and y. In this case τ is approximately normally distributed, with zero expectation value and a variance of

$$\text{Var}(\tau) = \frac{4N + 10}{9N(N-1)} \tag{14.6.9}$$

The following program proceeds according to the above description, and therefore loops over all pairs of data points. Beware: This is an $O(N^2)$ algorithm, unlike the algorithm for r_s, whose dominant sort operations are of order $N \log N$. If you are routinely computing Kendall's τ for data sets of more than a few thousand points, you may be in for some serious computing. If, however, you are willing to bin your data into a moderate number of bins, then read on.

```
      SUBROUTINE kendl1(data1,data2,n,tau,z,prob)
      INTEGER n
      REAL prob,tau,z,data1(n),data2(n)
C     USES erfcc
        Given data arrays data1(1:n) and data2(1:n), this program returns Kendall's τ as tau,
        its number of standard deviations from zero as z, and its two-sided significance level as prob.
        Small values of prob indicate a significant correlation (tau positive) or anticorrelation (tau
        negative).
      INTEGER is,j,k,n1,n2
      REAL a1,a2,aa,var,erfcc
      n1=0                                This will be the argument of one square root in (14.6.8),
      n2=0                                and this the other.
      is=0                                This will be the numerator in (14.6.8).
      do 12 j=1,n-1                       Loop over first member of pair,
        do 11 k=j+1,n                     and second member.
          a1=data1(j)-data1(k)
          a2=data2(j)-data2(k)
          aa=a1*a2
          if(aa.ne.0.)then                Neither array has a tie.
            n1=n1+1
            n2=n2+1
            if(aa.gt.0.)then
              is=is+1
            else
              is=is-1
            endif
          else                            One or both arrays have ties.
            if(a1.ne.0.)n1=n1+1                 An "extra x" event.
            if(a2.ne.0.)n2=n2+1                 An "extra y" event.
          endif
        enddo 11
      enddo 12
      tau=float(is)/sqrt(float(n1)*float(n2))   Equation (14.6.8).
      var=(4.*n+10.)/(9.*n*(n-1.))              Equation (14.6.9).
      z=tau/sqrt(var)
      prob=erfcc(abs(z)/1.4142136)              Significance.
      return
      END
```

Sometimes it happens that there are only a few possible values each for x and y. In that case, the data can be recorded as a contingency table (see §14.4) that gives the number of data points for each contingency of x and y.

Spearman's rank-order correlation coefficient is not a very natural statistic under these circumstances, since it assigns to each x and y bin a not-very-meaningful midrank value and then totals up vast numbers of identical rank differences. Kendall's tau, on the other hand, with its simple counting, remains quite natural. Furthermore, its $O(N^2)$ algorithm is no longer a problem, since we can arrange for it to loop over pairs of contingency table entries (each containing many data points) instead of over pairs of data points. This is implemented in the program that follows.

Note that Kendall's tau can be applied only to contingency tables where both variables are *ordinal*, i.e., well-ordered, and that it looks specifically for monotonic correlations, not for arbitrary associations. These two properties make it less general than the methods of §14.4, which applied to *nominal*, i.e., unordered, variables and arbitrary associations.

Comparing `kendl1` above with `kendl2` below, you will see that we have "floated" a number of variables. This is because the number of events in a contingency table might be sufficiently large as to cause overflows in some of the

integer arithmetic, while the number of individual data points in a list could not possibly be that large [for an $O(N^2)$ routine!].

```
      SUBROUTINE kendl2(tab,i,j,ip,jp,tau,z,prob)
      INTEGER i,ip,j,jp
      REAL prob,tau,z,tab(ip,jp)
C     USES erfcc
        Given a two-dimensional table tab of physical dimension (ip,jp) and logical dimension
        (i,j), such that tab(k,l) contains the number of events falling in bin k of one variable
        and bin l of another, this program returns Kendall's τ as tau, its number of standard
        deviations from zero as z, and its two-sided significance level as prob. Small values of prob
        indicate a significant correlation (tau positive) or anticorrelation (tau negative) between
        the two variables. Although tab is a real array, it will normally contain integral values.
      INTEGER k,ki,kj,l,li,lj,m1,m2,mm,nn
      REAL en1,en2,pairs,points,s,var,erfcc
      en1=0.                                  See kendl1 above.
      en2=0.
      s=0.
      nn=i*j                                  Total number of entries in contingency table.
      points=tab(i,j)
      do 12 k=0,nn-2                          Loop over entries in table,
        ki=k/j                                decoding a row index,
        kj=k-j*ki                             and a column index.
        points=points+tab(ki+1,kj+1)          Increment the total count of events.
        do 11 l=k+1,nn-1                      Loop over other member of the pair,
          li=l/j                              decoding its row
          lj=l-j*li                           and column.
          m1=li-ki
          m2=lj-kj
          mm=m1*m2
          pairs=tab(ki+1,kj+1)*tab(li+1,lj+1)
          if(mm.ne.0)then                     Not a tie.
            en1=en1+pairs
            en2=en2+pairs
            if(mm.gt.0)then                   Concordant, or
              s=s+pairs
            else                              discordant.
              s=s-pairs
            endif
          else
            if(m1.ne.0)en1=en1+pairs
            if(m2.ne.0)en2=en2+pairs
          endif
        enddo 11
      enddo 12
      tau=s/sqrt(en1*en2)
      var=(4.*points+10.)/(9.*points*(points-1.))
      z=tau/sqrt(var)
      prob=erfcc(abs(z)/1.4142136)
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Lehmann, E.L. 1975, *Nonparametrics: Statistical Methods Based on Ranks* (San Francisco: Holden-Day).

Downie, N.M., and Heath, R.W. 1965, *Basic Statistical Methods*, 2nd ed. (New York: Harper & Row), pp. 206–209.

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill).

14.7 Do Two-Dimensional Distributions Differ?

We here discuss a useful generalization of the K–S test (§14.3) to *two-dimensional* distributions. This generalization is due to Fasano and Franceschini [1], a variant on an earlier idea due to Peacock [2].

In a two-dimensional distribution, each data point is characterized by an (x, y) pair of values. An example near to our hearts is that each of the 19 neutrinos that were detected from Supernova 1987A is characterized by a time t_i and by an energy E_i (see [3]). We might wish to know whether these measured pairs (t_i, E_i), $i = 1 \ldots 19$ are consistent with a theoretical model that predicts neutrino flux as a function of both time and energy — that is, a two-dimensional probability distribution in the (x, y) [here, (t, E)] plane. That would be a one-sample test. Or, given two sets of neutrino detections, from two comparable detectors, we might want to know whether they are compatible with each other, a two-sample test.

In the spirit of the tried-and-true, one-dimensional K–S test, we want to range over the (x, y) plane in search of some kind of maximum *cumulative* difference between two two-dimensional distributions. Unfortunately, cumulative probability distribution is not well-defined in more than one dimension! Peacock's insight was that a good surrogate is the *integrated probability in each of four natural quadrants* around a given point (x_i, y_i), namely the total probabilities (or fraction of data) in $(x > x_i, y > y_i)$, $(x < x_i, y > y_i)$, $(x < x_i, y < y_i)$, $(x > x_i, y < y_i)$. The two-dimensional K–S statistic D is now taken to be the maximum difference (ranging both over data points and over quadrants) of the corresponding integrated probabilities. When comparing two data sets, the value of D may depend on which data set is ranged over. In that case, define an effective D as the average of the two values obtained. If you are confused at this point about the exact definition of D, don't fret; the accompanying computer routines amount to a precise algorithmic definition.

Figure 14.7.1 gives a feeling for what is going on. The 65 triangles and 35 squares seem to have somewhat different distributions in the plane. The dotted lines are centered on the triangle that maximizes the D statistic; the maximum occurs in the upper-left quadrant. That quadrant contains only 0.12 of all the triangles, but it contains 0.56 of all the squares. The value of D is thus 0.44. Is this statistically significant?

Even for fixed sample sizes, it is unfortunately not rigorously true that the distribution of D in the null hypothesis is independent of the shape of the two-dimensional distribution. In this respect the two-dimensional K–S test is not as natural as its one-dimensional parent. However, extensive Monte Carlo integrations have shown that the distribution of the two-dimensional D is *very nearly* identical for even quite different distributions, as long as they have the same coefficient of correlation r, defined in the usual way by equation (14.5.1). In their paper, Fasano and Franceschini tabulate Monte Carlo results for (what amounts to) the distribution of D as a function of (of course) D, sample size N, and coefficient of correlation r. Analyzing their results, one finds that the significance levels for the two-dimensional K–S test can be summarized by the simple, though approximate, formulas,

$$\text{Probability } (D > \text{observed}) = Q_{KS}\left(\frac{\sqrt{N}\,D}{1+\sqrt{1-r^2}(0.25-0.75/\sqrt{N})}\right) \tag{14.7.1}$$

for the one-sample case, and the same for the two-sample case, but with

$$N = \frac{N_1 N_2}{N_1 + N_2}. \tag{14.7.2}$$

The above formulas are accurate enough when $N \gtrsim 20$, and when the indicated probability (significance level) is less than (more significant than) 0.20 or so. When the indicated probability is > 0.20, its value may not be accurate, but the implication that the data and model (or two data sets) are not significantly different is certainly correct. Notice that in the limit of $r \to 1$ (perfect correlation), equations (14.7.1) and (14.7.2) reduce to equations (14.3.9) and (14.3.10): The two-dimensional data lie on a perfect straight line, and the two-dimensional K–S test becomes a one-dimensional K–S test.

14.7 Do Two-Dimensional Distributions Differ?

We here discuss a useful generalization of the K–S test (§14.3) to *two-dimensional* distributions. This generalization is due to Fasano and Franceschini [1], a variant on an earlier idea due to Peacock [2].

In a two-dimensional distribution, each data point is characterized by an (x, y) pair of values. An example near to our hearts is that each of the 19 neutrinos that were detected from Supernova 1987A is characterized by a time t_i and by an energy E_i (see [3]). We might wish to know whether these measured pairs (t_i, E_i), $i = 1 \ldots 19$ are consistent with a theoretical model that predicts neutrino flux as a function of both time and energy — that is, a two-dimensional probability distribution in the (x, y) [here, (t, E)] plane. That would be a one-sample test. Or, given two sets of neutrino detections, from two comparable detectors, we might want to know whether they are compatible with each other, a two-sample test.

In the spirit of the tried-and-true, one-dimensional K–S test, we want to range over the (x, y) plane in search of some kind of maximum *cumulative* difference between two two-dimensional distributions. Unfortunately, cumulative probability distribution is not well-defined in more than one dimension! Peacock's insight was that a good surrogate is the *integrated probability in each of four natural quadrants* around a given point (x_i, y_i), namely the total probabilities (or fraction of data) in $(x > x_i, y > y_i)$, $(x < x_i, y > y_i)$, $(x < x_i, y < y_i)$, $(x > x_i, y < y_i)$. The two-dimensional K–S statistic D is now taken to be the maximum difference (ranging both over data points and over quadrants) of the corresponding integrated probabilities. When comparing two data sets, the value of D may depend on which data set is ranged over. In that case, define an effective D as the average of the two values obtained. If you are confused at this point about the exact definition of D, don't fret; the accompanying computer routines amount to a precise algorithmic definition.

Figure 14.7.1 gives a feeling for what is going on. The 65 triangles and 35 squares seem to have somewhat different distributions in the plane. The dotted lines are centered on the triangle that maximizes the D statistic; the maximum occurs in the upper-left quadrant. That quadrant contains only 0.12 of all the triangles, but it contains 0.56 of all the squares. The value of D is thus 0.44. Is this statistically significant?

Even for fixed sample sizes, it is unfortunately not rigorously true that the distribution of D in the null hypothesis is independent of the shape of the two-dimensional distribution. In this respect the two-dimensional K–S test is not as natural as its one-dimensional parent. However, extensive Monte Carlo integrations have shown that the distribution of the two-dimensional D is *very nearly* identical for even quite different distributions, as long as they have the same coefficient of correlation r, defined in the usual way by equation (14.5.1). In their paper, Fasano and Franceschini tabulate Monte Carlo results for (what amounts to) the distribution of D as a function of (of course) D, sample size N, and coefficient of correlation r. Analyzing their results, one finds that the significance levels for the two-dimensional K–S test can be summarized by the simple, though approximate, formulas,

$$\text{Probability } (D > \text{observed}) = Q_{KS}\left(\frac{\sqrt{N}\,D}{1+\sqrt{1-r^2}(0.25-0.75/\sqrt{N})}\right) \tag{14.7.1}$$

for the one-sample case, and the same for the two-sample case, but with

$$N = \frac{N_1 N_2}{N_1 + N_2}. \tag{14.7.2}$$

The above formulas are accurate enough when $N \gtrsim 20$, and when the indicated probability (significance level) is less than (more significant than) 0.20 or so. When the indicated probability is > 0.20, its value may not be accurate, but the implication that the data and model (or two data sets) are not significantly different is certainly correct. Notice that in the limit of $r \to 1$ (perfect correlation), equations (14.7.1) and (14.7.2) reduce to equations (14.3.9) and (14.3.10): The two-dimensional data lie on a perfect straight line, and the two-dimensional K–S test becomes a one-dimensional K–S test.

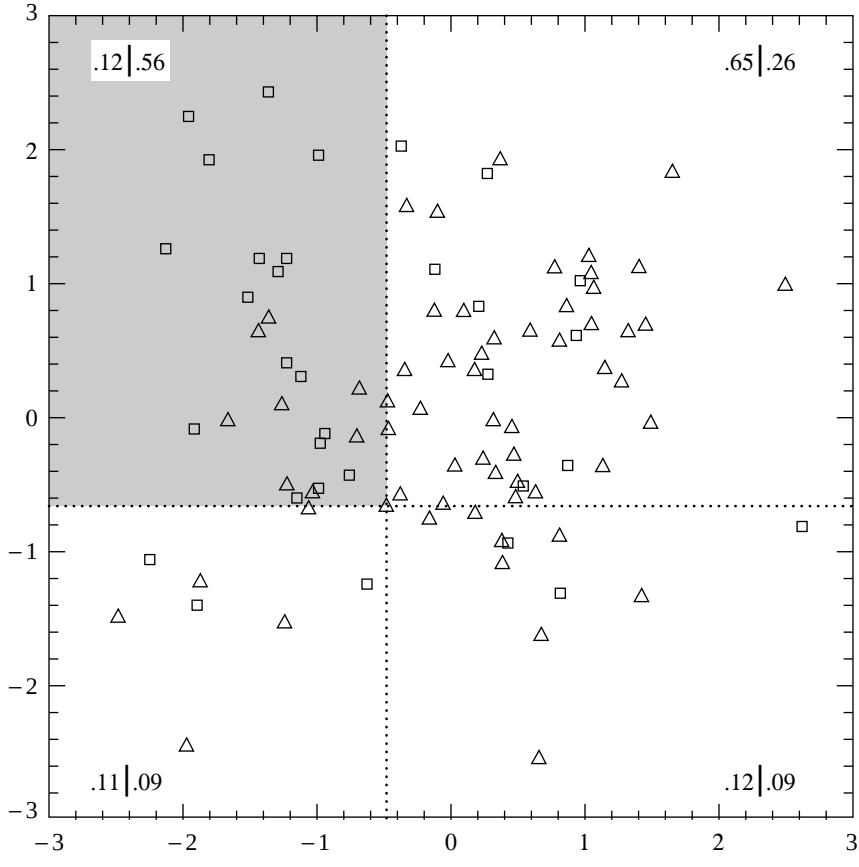


Figure 14.7.1. Two-dimensional distributions of 65 triangles and 35 squares. The two-dimensional K–S test finds that point one of whose quadrants (shown by dotted lines) maximizes the difference between fraction of triangles and fraction of squares. Then, equation (14.7.1) indicates whether the difference is statistically significant, i.e., whether the triangles and squares must have different underlying distributions.

The significance level for the data in Figure 14.7.1, by the way, is about 0.001. This establishes to a near-certainty that the triangles and squares were drawn from different distributions. (As in fact they were.)

Of course, if you do not want to rely on the Monte Carlo experiments embodied in equation (14.7.1), you can do your own: Generate a lot of synthetic data sets from your model, each one with the same number of points as the real data set. Compute D for each synthetic data set, using the accompanying computer routines (but ignoring their calculated probabilities), and count what fraction of the time these synthetic D's exceed the D from the real data. That fraction is your significance.

One disadvantage of the two-dimensional tests, by comparison with their one-dimensional progenitors, is that the two-dimensional tests require of order N^2 operations: Two nested loops of order N take the place of an $N \log N$ sort. For small computers, this restricts the usefulness of the tests to N less than several thousand.

We now give computer implementations. The one-sample case is embodied in the routine `ks2d1s` (that is, 2-dimensions, 1-sample). This routine calls a straightforward utility routine `quadct` to count points in the four quadrants, and it calls a user-supplied routine `quadvl` that must be capable of returning the integrated probability of an analytic model in each of four quadrants around an arbitrary (x, y) point. A trivial sample `quadvl` is shown; realistic `quadvl`s can be quite complicated, often incorporating numerical quadratures over

analytic two-dimensional distributions.

```
      SUBROUTINE ks2d1s(x1,y1,n1,quadvl,d1,prob)
      INTEGER n1
      REAL d1,prob,x1(n1),y1(n1)
      EXTERNAL quadvl
C     USES pearsn,probks,quadct,quadvl
```

Two-dimensional Kolmogorov-Smirnov test of one sample against a model. Given the x and y coordinates of `n1` data points in arrays `x1(1:n1)` and `y1(1:n1)`, and given a user-supplied function `quadvl` that exemplifies the model, this routine returns the two-dimensional K-S statistic as `d1`, and its significance level as `prob`. Small values of `prob` show that the sample is significantly different from the model. Note that the test is slightly distribution-dependent, so `prob` is only an estimate.

```
      INTEGER j
      REAL dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,rr,sqen,probks
      d1=0.0
      do 11 j=1,n1                              Loop over the data points.
        call quadct(x1(j),y1(j),x1,y1,n1,fa,fb,fc,fd)
        call quadvl(x1(j),y1(j),ga,gb,gc,gd)
        d1=max(d1,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
```

For both the sample and the model, the distribution is integrated in each of four quadrants, and the maximum difference is saved.

```
      enddo 11
      call pearsn(x1,y1,n1,r1,dum,dumm)  Get the linear correlation coefficient r1.
      sqen=sqrt(float(n1))
      rr=sqrt(1.0-r1**2)
```

Estimate the probability using the K-S probability function `probks`.

```
      prob=probks(d1*sqen/(1.0+rr*(0.25-0.75/sqen)))
      return
      END
```

```
      SUBROUTINE quadct(x,y,xx,yy,nn,fa,fb,fc,fd)
      INTEGER nn
      REAL fa,fb,fc,fd,x,y,xx(nn),yy(nn)
```

Given an origin (x, y), and an array of `nn` points with coordinates `xx` and `yy`, count how many of them are in each quadrant around the origin, and return the normalized fractions. Quadrants are labeled alphabetically, counterclockwise from the upper right. Used by `ks2d1s` and `ks2d2s`.

```
      INTEGER k,na,nb,nc,nd
      REAL ff
      na=0
      nb=0
      nc=0
      nd=0
      do 11 k=1,nn
        if(yy(k).gt.y)then
          if(xx(k).gt.x)then
            na=na+1
          else
            nb=nb+1
          endif
        else
          if(xx(k).gt.x)then
            nd=nd+1
          else
            nc=nc+1
          endif
        endif
      enddo 11
      ff=1.0/nn
      fa=ff*na
      fb=ff*nb
```

```
      fc=ff*nc
      fd=ff*nd
      return
      END



      SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
      REAL fa,fb,fc,fd,x,y
          This is a sample of a user-supplied routine to be used with ks2d1s. In this case, the model
          distribution is uniform inside the square −1 < x < 1, −1 < y < 1. In general this routine
          should return, for any point (x, y), the fraction of the total distribution in each of the
          four quadrants around that point. The fractions, fa, fb, fc, and fd, must add up to 1.
          Quadrants are alphabetical, counterclockwise from the upper right.
      REAL qa,qb,qc,qd
      qa=min(2.,max(0.,1.-x))
      qb=min(2.,max(0.,1.-y))
      qc=min(2.,max(0.,x+1.))
      qd=min(2.,max(0.,y+1.))
      fa=0.25*qa*qb
      fb=0.25*qb*qc
      fc=0.25*qc*qd
      fd=0.25*qd*qa
      return
      END
```

The routine ks2d2s is the two-sample case of the two-dimensional K–S test. It also calls quadct, pearsn, and probks. Being a two-sample test, it does not need an analytic model.

```
      SUBROUTINE ks2d2s(x1,y1,n1,x2,y2,n2,d,prob)
      INTEGER n1,n2
      REAL d,prob,x1(n1),x2(n2),y1(n1),y2(n2)
C     USES pearsn,probks,quadct
          Two-dimensional Kolmogorov-Smirnov test on two samples. Given the x and y coordinates
          of the first sample as n1 values in arrays x1(1:n1) and y1(1:n1), and likewise for the
          second sample, n2 values in arrays x2 and y2, this routine returns the two-dimensional, two-
          sample K-S statistic as d, and its significance level as prob. Small values of prob show
          that the two samples are significantly different. Note that the test is slightly distribution-
          dependent, so prob is only an estimate.
      INTEGER j
      REAL d1,d2,dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,r2,rr,
     *     sqen,probks
      d1=0.0
      do 11 j=1,n1                          First, use points in the first sample as origins.
          call quadct(x1(j),y1(j),x1,y1,n1,fa,fb,fc,fd)
          call quadct(x1(j),y1(j),x2,y2,n2,ga,gb,gc,gd)
          d1=max(d1,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
      enddo 11
      d2=0.0
      do 12 j=1,n2                          Then, use points in the second sample as origins.
          call quadct(x2(j),y2(j),x1,y1,n1,fa,fb,fc,fd)
          call quadct(x2(j),y2(j),x2,y2,n2,ga,gb,gc,gd)
          d2=max(d2,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
      enddo 12
      d=0.5*(d1+d2)                         Average the K-S statistics.
      sqen=sqrt(float(n1)*float(n2)/float(n1+n2))
      call pearsn(x1,y1,n1,r1,dum,dumm)     Get the linear correlation coefficient for each sample.
      call pearsn(x2,y2,n2,r2,dum,dumm)
      rr=sqrt(1.0-0.5*(r1**2+r2**2))
        Estimate the probability using the K-S probability function probks.
      prob=probks(d*sqen/(1.0+rr*(0.25-0.75/sqen)))
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Fasano, G. and Franceschini, A. 1987, *Monthly Notices of the Royal Astronomical Society*, vol. 225, pp. 155–170. [1]

Peacock, J.A. 1983, *Monthly Notices of the Royal Astronomical Society*, vol. 202, pp. 615–627. [2]

Spergel, D.N., Piran, T., Loeb, A., Goodman, J., and Bahcall, J.N. 1987, *Science*, vol. 237, pp. 1471–1473. [3]

14.8 Savitzky-Golay Smoothing Filters

In §13.5 we learned something about the construction and application of digital filters, but little guidance was given on *which particular* filter to use. That, of course, depends on what you want to accomplish by filtering. One obvious use for *low-pass* filters is to smooth noisy data.

The premise of data smoothing is that one is measuring a variable that is both slowly varying and also corrupted by random noise. Then it can sometimes be useful to replace each data point by some kind of local average of surrounding data points. Since nearby points measure very nearly the same underlying value, averaging can reduce the level of noise without (much) biasing the value obtained.

We must comment editorially that the smoothing of data lies in a murky area, beyond the fringe of some better posed, and therefore more highly recommended, techniques that are discussed elsewhere in this book. If you are fitting data to a parametric model, for example (see Chapter 15), it is almost always better to use raw data than to use data that has been pre-processed by a smoothing procedure. Another alternative to blind smoothing is so-called "optimal" or Wiener filtering, as discussed in §13.3 and more generally in §13.6. Data smoothing is probably most justified when it is used simply as a graphical technique, to guide the eye through a forest of data points all with large error bars; or as a means of making initial *rough* estimates of simple parameters from a graph.

In this section we discuss a particular type of low-pass filter, well-adapted for data smoothing, and termed variously *Savitzky-Golay* [1], *least-squares* [2], or *DISPO* (Digital Smoothing Polynomial) [3] filters. Rather than having their properties defined in the Fourier domain, and then translated to the time domain, Savitzky-Golay filters derive directly from a particular formulation of the data smoothing problem in the time domain, as we will now see. Savitzky-Golay filters were initially (and are still often) used to render visible the relative widths and heights of spectral lines in noisy spectrometric data.

Recall that a digital filter is applied to a series of equally spaced data values $f_i \equiv f(t_i)$, where $t_i \equiv t_0 + i\Delta$ for some constant sample spacing Δ and $i = \ldots -2, -1, 0, 1, 2, \ldots$. We have seen (§13.5) that the simplest type of digital filter (the nonrecursive or finite impulse response filter) replaces each data value f_i by a linear combination g_i of itself and some number of nearby neighbors,

$$g_i = \sum_{n=-n_L}^{n_R} c_n f_{i+n} \tag{14.8.1}$$

Here n_L is the number of points used "to the left" of a data point i, i.e., earlier than it, while n_R is the number used to the right, i.e., later. A so-called *causal* filter would have $n_R = 0$.

As a starting point for understanding Savitzky-Golay filters, consider the simplest possible averaging procedure: For some fixed $n_L = n_R$, compute each g_i as the average of the data points from f_{i-n_L} to f_{i+n_R}. This is sometimes called *moving window averaging* and corresponds to equation (14.8.1) with constant $c_n = 1/(n_L + n_R + 1)$. If the underlying function is constant, or is changing linearly with time (increasing or decreasing), then no bias is introduced into the result. Higher points at one end of the averaging interval are on

CITED REFERENCES AND FURTHER READING:

Fasano, G. and Franceschini, A. 1987, *Monthly Notices of the Royal Astronomical Society*, vol. 225, pp. 155–170. [1]

Peacock, J.A. 1983, *Monthly Notices of the Royal Astronomical Society*, vol. 202, pp. 615–627. [2]

Spergel, D.N., Piran, T., Loeb, A., Goodman, J., and Bahcall, J.N. 1987, *Science*, vol. 237, pp. 1471–1473. [3]

14.8 Savitzky-Golay Smoothing Filters

In §13.5 we learned something about the construction and application of digital filters, but little guidance was given on *which particular* filter to use. That, of course, depends on what you want to accomplish by filtering. One obvious use for *low-pass* filters is to smooth noisy data.

The premise of data smoothing is that one is measuring a variable that is both slowly varying and also corrupted by random noise. Then it can sometimes be useful to replace each data point by some kind of local average of surrounding data points. Since nearby points measure very nearly the same underlying value, averaging can reduce the level of noise without (much) biasing the value obtained.

We must comment editorially that the smoothing of data lies in a murky area, beyond the fringe of some better posed, and therefore more highly recommended, techniques that are discussed elsewhere in this book. If you are fitting data to a parametric model, for example (see Chapter 15), it is almost always better to use raw data than to use data that has been pre-processed by a smoothing procedure. Another alternative to blind smoothing is so-called "optimal" or Wiener filtering, as discussed in §13.3 and more generally in §13.6. Data smoothing is probably most justified when it is used simply as a graphical technique, to guide the eye through a forest of data points all with large error bars; or as a means of making initial *rough* estimates of simple parameters from a graph.

In this section we discuss a particular type of low-pass filter, well-adapted for data smoothing, and termed variously *Savitzky-Golay* [1], *least-squares* [2], or *DISPO* (Digital Smoothing Polynomial) [3] filters. Rather than having their properties defined in the Fourier domain, and then translated to the time domain, Savitzky-Golay filters derive directly from a particular formulation of the data smoothing problem in the time domain, as we will now see. Savitzky-Golay filters were initially (and are still often) used to render visible the relative widths and heights of spectral lines in noisy spectrometric data.

Recall that a digital filter is applied to a series of equally spaced data values $f_i \equiv f(t_i)$, where $t_i \equiv t_0 + i\Delta$ for some constant sample spacing Δ and $i = \ldots -2, -1, 0, 1, 2, \ldots$. We have seen (§13.5) that the simplest type of digital filter (the nonrecursive or finite impulse response filter) replaces each data value f_i by a linear combination g_i of itself and some number of nearby neighbors,

$$g_i = \sum_{n=-n_L}^{n_R} c_n f_{i+n} \tag{14.8.1}$$

Here n_L is the number of points used "to the left" of a data point i, i.e., earlier than it, while n_R is the number used to the right, i.e., later. A so-called *causal* filter would have $n_R = 0$.

As a starting point for understanding Savitzky-Golay filters, consider the simplest possible averaging procedure: For some fixed $n_L = n_R$, compute each g_i as the average of the data points from f_{i-n_L} to f_{i+n_R}. This is sometimes called *moving window averaging* and corresponds to equation (14.8.1) with constant $c_n = 1/(n_L + n_R + 1)$. If the underlying function is constant, or is changing linearly with time (increasing or decreasing), then no bias is introduced into the result. Higher points at one end of the averaging interval are on

the average balanced by lower points at the other end. A bias is introduced, however, if the underlying function has a nonzero second derivative. At a local maximum, for example, moving window averaging always reduces the function value. In the spectrometric application, a narrow spectral line has its height reduced and its width increased. Since these parameters are themselves of physical interest, the bias introduced is distinctly undesirable.

Note, however, that moving window averaging does preserve the area under a spectral line, which is its zeroth moment, and also (if the window is symmetric with $n_L = n_R$) its mean position in time, which is its first moment. What is violated is the second moment, equivalent to the line width.

The idea of Savitzky-Golay filtering is to find filter coefficients c_n that preserve higher moments. Equivalently, the idea is to approximate the underlying function within the moving window not by a constant (whose estimate is the average), but by a polynomial of higher order, typically quadratic or quartic: For each point f_i, we least-squares fit a polynomial to all $n_L + n_R + 1$ points in the moving window, and then set g_i to be the value of that polynomial at position i. (If you are not familiar with least-squares fitting, you might want to look ahead to Chapter 15.) We make no use of the value of the polynomial at any other point. When we move on to the next point f_{i+1}, we do a whole new least-squares fit using a shifted window.

All these least-squares fits would be laborious if done as described. Luckily, since the process of least-squares fitting involves only a linear matrix inversion, the coefficients of a fitted polynomial are themselves linear in the values of the data. That means that we can do all the fitting in advance, for fictitious data consisting of all zeros except for a single 1, and then do the fits on the real data just by taking linear combinations. This is the key point, then: There are particular sets of filter coefficients c_n for which equation (14.8.1) "automatically" accomplishes the process of polynomial least-squares fitting inside a moving window.

To derive such coefficients, consider how g_0 might be obtained: We want to fit a polynomial of degree M in i, namely $a_0 + a_1 i + \cdots + a_M i^M$ to the values $f_{-n_L}, \ldots, f_{n_R}$. Then g_0 will be the value of that polynomial at $i = 0$, namely a_0. The design matrix for this problem (§15.4) is

$$A_{ij} = i^j \qquad i = -n_L, \ldots, n_R, \quad j = 0, \ldots, M \tag{14.8.2}$$

and the normal equations for the vector of a_j's in terms of the vector of f_i's is in matrix notation

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{a} = \mathbf{A}^T \cdot \mathbf{f} \qquad \text{or} \qquad \mathbf{a} = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^T \cdot \mathbf{f}) \tag{14.8.3}$$

We also have the specific forms

$$\left\{\mathbf{A}^T \cdot \mathbf{A}\right\}_{ij} = \sum_{k=-n_L}^{n_R} A_{ki} A_{kj} = \sum_{k=-n_L}^{n_R} k^{i+j} \tag{14.8.4}$$

and

$$\left\{\mathbf{A}^T \cdot \mathbf{f}\right\}_{j} = \sum_{k=-n_L}^{n_R} A_{kj} f_k = \sum_{k=-n_L}^{n_R} k^j f_k \tag{14.8.5}$$

Since the coefficient c_n is the component a_0 when $\mathbf{f}$ is replaced by the unit vector $\mathbf{e}_n$, $-n_L \le n < n_R$, we have

$$c_n = \left\{(\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^T \cdot \mathbf{e}_n)\right\}_0 = \sum_{m=0}^{M} \left\{(\mathbf{A}^T \cdot \mathbf{A})^{-1}\right\}_{0m} n^m \tag{14.8.6}$$

Note that equation (14.8.6) says that we need only one row of the inverse matrix. (Numerically we can get this by LU decomposition with only a single backsubstitution.)

The subroutine `savgol`, below, implements equation (14.8.6). As input, it takes the parameters `nl` $= n_L$, `nr` $= n_R$, and `m` $= M$ (the desired order). Also input is `np`, the physical length of the output array `c`, and a parameter `ld` which for data fitting should be zero. In fact, `ld` specifies which coefficient among the a_i's should be returned, and we are here interested in a_0. For another purpose, namely the computation of numerical derivatives (already mentioned in §5.7) the useful choice is `ld` ≥ 1. With `ld` $= 1$, for example, the filtered first derivative is the convolution (14.8.1) divided by the stepsize Δ. For derivatives, one usually wants `m` $= 4$ or larger.

M	n_L	n_R	Sample Savitzky-Golay Coefficients										
2	2	2				−0.086	0.343	0.486	0.343	−0.086			
2	3	1			−0.143	0.171	0.343	0.371	0.257				
2	4	0		0.086	−0.143	−0.086	0.257	0.886					
2	5	5	−0.084	0.021	0.103	0.161	0.196	0.207	0.196	0.161	0.103	0.021	−0.084
4	4	4		0.035	−0.128	0.070	0.315	0.417	0.315	0.070	−0.128	0.035	
4	5	5	0.042	−0.105	−0.023	0.140	0.280	0.333	0.280	0.140	−0.023	−0.105	0.042

```
      SUBROUTINE savgol(c,np,nl,nr,ld,m)
      INTEGER ld,m,nl,np,nr,MMAX
      REAL c(np)
      PARAMETER (MMAX=6)
C     USES lubksb,ludcmp
        Returns in c(1:np), in wrap-around order (N.B.!) consistent with the argument respns
        in routine convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward
        (past) data points used, while nr is the number of rightward (future) data points, making
        the total number of data points used nl + nr + 1. ld is the order of the derivative desired
        (e.g., ld = 0 for smoothed function). m is the order of the smoothing polynomial, also
        equal to the highest conserved moment; usual values are m = 2 or m = 4.
      INTEGER imj,ipj,j,k,kk,mm,indx(MMAX+1)
      REAL d,fac,sum,a(MMAX+1,MMAX+1),b(MMAX+1)
      if(np.lt.nl+nr+1.or.nl.lt.0.or.nr.lt.0.or.ld.gt.m.or.m.gt.MMAX
     *    .or.nl+nr.lt.m) pause 'bad args in savgol'
      do 14 ipj=0,2*m                          Set up the normal equations of the desired least-
        sum=0.                                     squares fit.
        if(ipj.eq.0)sum=1.
        do 11 k=1,nr
          sum=sum+float(k)**ipj
        enddo 11
        do 12 k=1,nl
          sum=sum+float(-k)**ipj
        enddo 12
        mm=min(ipj,2*m-ipj)
        do 13 imj=-mm,mm,2
          a(1+(ipj+imj)/2,1+(ipj-imj)/2)=sum
        enddo 13
      enddo 14
      call ludcmp(a,m+1,MMAX+1,indx,d)         Solve them: LU decomposition.
      do 15 j=1,m+1
        b(j)=0.
      enddo 15
      b(ld+1)=1.      Right-hand side vector is unit vector, depending on which derivative we want.
      call lubksb(a,m+1,MMAX+1,indx,b)         Backsubstitute, giving one row of the inverse matrix.
      do 16 kk=1,np                            Zero the output array (it may be bigger than number
        c(kk)=0.                                   of coefficients).
      enddo 16
      do 18 k=-nl,nr                           Each Savitzky-Golay coefficient is the dot product
        sum=b(1)                                   of powers of an integer with the inverse matrix
        fac=1.                                     row.
        do 17 mm=1,m
          fac=fac*k
          sum=sum+b(mm+1)*fac
        enddo 17
        kk=mod(np-k,np)+1                      Store in wrap-around order.
        c(kk)=sum
      enddo 18
      return
      END
```

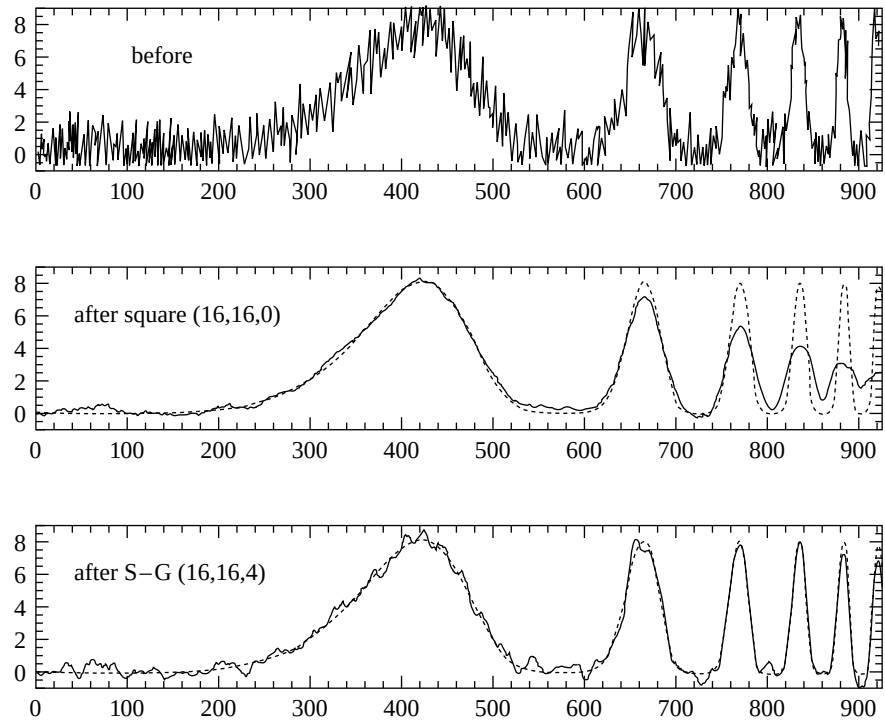


Figure 14.8.1. Top: Synthetic noisy data consisting of a sequence of progressively narrower bumps, and additive Gaussian white noise. Center: Result of smoothing the data by a simple moving window average. The window extends 16 points leftward and rightward, for a total of 33 points. Note that narrow features are broadened and suffer corresponding loss of amplitude. The dotted curve is the underlying function used to generate the synthetic data. Bottom: Result of smoothing the data by a Savitzky-Golay smoothing filter (of degree 4) using the same 33 points. While there is less smoothing of the broadest feature, narrower features have their heights and widths preserved.

As output, `savgol` returns the coefficients c_n, for $-n_L \le n \le n_R$. These are stored in `c` in "wrap-around order"; that is, c_0 is in `c(1)`, c_{-1} is in `c(2)`, and so on for further negative indices. The value c_1 is stored in `c(np)`, c_2 in `c(np-1)`, and so on for positive indices. This order may seem arcane, but it is the natural one where causal filters have nonzero coefficients in low array elements of `c`. It is also the order required by the subroutine `convlv` in §13.1, which can be used to apply the digital filter to a data set.

The accompanying table shows some typical output from `savgol`. For orders 2 and 4, the coefficients of Savitzky-Golay filters with several choices of n_L and n_R are shown. The central column is the coefficient applied to the data f_i in obtaining the smoothed g_i. Coefficients to the left are applied to earlier data; to the right, to later. The coefficients always add (within roundoff error) to unity. One sees that, as befits a smoothing operator, the coefficients always have a central positive lobe, but with smaller, outlying corrections of both positive and negative sign. In practice, the Savitzky-Golay filters are most useful for much larger values of n_L and n_R, since these few-point formulas can accomplish only a relatively small amount of smoothing.

Figure 14.8.1 shows a numerical experiment using a 33 point smoothing filter, that is, $n_L = n_R = 16$. The upper panel shows a test function, constructed to have six "bumps" of varying widths, all of height 8 units. To this function Gaussian white noise of unit variance has been added. (The test function without noise is shown as the dotted curves in the center and lower panels.) The widths of the bumps (full width at half of maximum, or FWHM) are 140, 43, 24, 17, 13, and 10, respectively.

The middle panel of Figure 14.8.1 shows the result of smoothing by a moving window

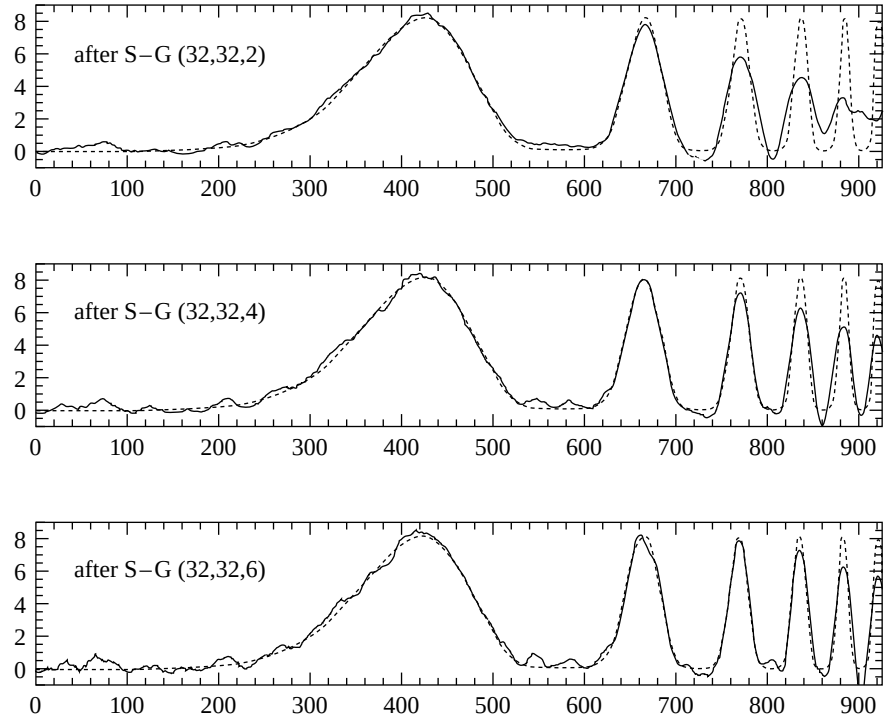


Figure 14.8.2. Result of applying wider 65 point Savitzky-Golay filters to the same data set as in Figure 14.8.1. Top: degree 2. Center: degree 4. Bottom: degree 6. All of these filters are inoptimally broad for the resolution of the narrow features. Higher-order filters do best at preserving feature heights and widths, but do less smoothing on broader features.

average. One sees that the window of width 33 does quite a nice job of smoothing the broadest bump, but that the narrower bumps suffer considerable loss of height and increase of width. The underlying signal (dotted) is very badly represented.

The lower panel shows the result of smoothing with a Savitzky-Golay filter of the identical width, and degree $M = 4$. One sees that the heights and widths of the bumps are quite extraordinarily preserved. A trade-off is that the broadest bump is less smoothed. That is because the central positive lobe of the Savitzky-Golay filter coefficients fills only a fraction of the full 33 point width. As a rough guideline, best results are obtained when the full width of the degree 4 Savitzky-Golay filter is between 1 and 2 times the FWHM of desired features in the data. (References [3] and [4] give additional practical hints.)

Figure 14.8.2 shows the result of smoothing the same noisy "data" with broader Savitzky-Golay filters of 3 different orders. Here we have $n_L = n_R = 32$ (65 point filter) and $M = 2, 4, 6$. One sees that, when the bumps are too narrow with respect to the filter size, then even the Savitzky-Golay filter must at some point give out. The higher order filter manages to track narrower features, but at the cost of less smoothing on broad features.

To summarize: Within limits, Savitzky-Golay filtering does manage to provide smoothing without loss of resolution. It does this by assuming that relatively distant data points have some significant redundancy that can be used to reduce the level of noise. The specific nature of the assumed redundancy is that the underlying function should be locally well-fitted by a polynomial. When this is true, as it is for smooth line profiles not too much narrower than the filter width, then the performance of Savitzky-Golay filters can be spectacular. When it is not true, then these filters have no compelling advantage over other classes of smoothing filter coefficients.

A last remark concerns irregularly sampled data, where the values f_i are not uniformly spaced in time. The obvious generalization of Savitzky-Golay filtering would be to do a

least-squares fit within a moving window around each data point, one containing a fixed number of data points to the left (n_L) and right (n_R). Because of the irregular spacing, however, there is no way to obtain universal filter coefficients applicable to more than one data point. One must instead do the actual least-squares fits for each data point. This becomes computationally burdensome for larger n_L, n_R, and M.

As a cheap alternative, one can simply pretend that the data points *are* equally spaced. This amounts to virtually shifting, within each moving window, the data points to equally spaced positions. Such a shift introduces the equivalent of an additional source of noise into the function values. In those cases where smoothing is useful, this noise will often be much smaller than the noise already present. Specifically, if the location of the points is approximately random within the window, then a rough criterion is this: If the change in f across the full width of the $N = n_L + n_R + 1$ point window is less than $\sqrt{N/2}$ times the measurement noise on a single point, then the cheap method can be used.

CITED REFERENCES AND FURTHER READING:

Savitzky A., and Golay, M.J.E. 1964, *Analytical Chemistry*, vol. 36, pp. 1627–1639. [1]

Hamming, R.W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall). [2]

Ziegler, H. 1981, *Applied Spectroscopy*, vol. 35, pp. 88–92. [3]

Bromba, M.U.A., and Ziegler, H. 1981, *Analytical Chemistry*, vol. 53, pp. 1583–1586. [4]

Chapter 15. Modeling of Data

15.0 Introduction

Given a set of observations, one often wants to condense and summarize the data by fitting it to a "model" that depends on adjustable parameters. Sometimes the model is simply a convenient class of functions, such as polynomials or Gaussians, and the fit supplies the appropriate coefficients. Other times, the model's parameters come from some underlying theory that the data are supposed to satisfy; examples are coefficients of rate equations in a complex network of chemical reactions, or orbital elements of a binary star. Modeling can also be used as a kind of constrained interpolation, where you want to extend a few data points into a continuous function, but with some underlying idea of what that function should look like.

The basic approach in all cases is usually the same: You choose or design a *figure-of-merit function* ("merit function," for short) that measures the agreement between the data and the model with a particular choice of parameters. The merit function is conventionally arranged so that small values represent close agreement. The parameters of the model are then adjusted to achieve a minimum in the merit function, yielding *best-fit parameters*. The adjustment process is thus a problem in minimization in many dimensions. This optimization was the subject of Chapter 10; however, there exist special, more efficient, methods that are specific to modeling, and we will discuss these in this chapter.

There are important issues that go beyond the mere finding of best-fit parameters. Data are generally not exact. They are subject to *measurement errors* (called *noise* in the context of signal-processing). Thus, typical data never exactly fit the model that is being used, even when that model is correct. We need the means to assess whether or not the model is appropriate, that is, we need to test the *goodness-of-fit* against some useful statistical standard.

We usually also need to know the accuracy with which parameters are determined by the data set. In other words, we need to know the likely errors of the best-fit parameters.

Finally, it is not uncommon in fitting data to discover that the merit function is not unimodal, with a single minimum. In some cases, we may be interested in global rather than local questions. Not, "how good is this fit?" but rather, "how sure am I that there is not a *very much better* fit in some corner of parameter space?" As we have seen in Chapter 10, especially §10.9, this kind of problem is generally quite difficult to solve.

The important message we want to deliver is that fitting of parameters is not the end-all of parameter estimation. To be genuinely useful, a fitting procedure

should provide (i) parameters, (ii) error estimates on the parameters, and (iii) a statistical measure of goodness-of-fit. When the third item suggests that the model is an unlikely match to the data, then items (i) and (ii) are probably worthless. Unfortunately, many practitioners of parameter estimation never proceed beyond item (i). They deem a fit acceptable if a graph of data and model "looks good." This approach is known as *chi-by-eye*. Luckily, its practitioners get what they deserve.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill).

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapter X.

Korn, G.A., and Korn, T.M. 1968, *Mathematical Handbook for Scientists and Engineers*, 2nd ed. (New York: McGraw-Hill), Chapters 18–19.

15.1 Least Squares as a Maximum Likelihood Estimator

Suppose that we are fitting N data points (x_i, y_i) $i = 1, \ldots, N$, to a model that has M adjustable parameters $a_j,\ j = 1, \ldots, M$. The model predicts a functional relationship between the measured independent and dependent variables,

$$y(x) = y(x; a_1 \ldots a_M) \tag{15.1.1}$$

where the dependence on the parameters is indicated explicitly on the right-hand side.

What, exactly, do we want to minimize to get fitted values for the a_j's? The first thing that comes to mind is the familiar least-squares fit,

$$\text{minimize over } a_1 \ldots a_M : \qquad \sum_{i=1}^{N} [y_i - y(x_i; a_1 \ldots a_M)]^2 \tag{15.1.2}$$

But where does this come from? What general principles is it based on? The answer to these questions takes us into the subject of *maximum likelihood estimators*.

Given a particular data set of x_i's and y_i's, we have the intuitive feeling that some parameter sets $a_1 \ldots a_M$ are very unlikely — those for which the model function $y(x)$ looks *nothing like* the data — while others may be very likely — those that closely resemble the data. How can we quantify this intuitive feeling? How can we select fitted parameters that are "most likely" to be correct? It is not meaningful to ask the question, "What is the probability that a particular set of fitted parameters $a_1 \ldots a_M$ is correct?" The reason is that there is no statistical universe of models from which the parameters are drawn. There is just one model, the correct one, and a statistical universe of data sets that are drawn from it!

should provide (i) parameters, (ii) error estimates on the parameters, and (iii) a statistical measure of goodness-of-fit. When the third item suggests that the model is an unlikely match to the data, then items (i) and (ii) are probably worthless. Unfortunately, many practitioners of parameter estimation never proceed beyond item (i). They deem a fit acceptable if a graph of data and model "looks good." This approach is known as *chi-by-eye*. Luckily, its practitioners get what they deserve.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill).

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapter X.

Korn, G.A., and Korn, T.M. 1968, *Mathematical Handbook for Scientists and Engineers*, 2nd ed. (New York: McGraw-Hill), Chapters 18–19.

15.1 Least Squares as a Maximum Likelihood Estimator

Suppose that we are fitting N data points (x_i, y_i) $i = 1, \ldots, N$, to a model that has M adjustable parameters $a_j,\ j = 1, \ldots, M$. The model predicts a functional relationship between the measured independent and dependent variables,

$$y(x) = y(x; a_1 \ldots a_M) \tag{15.1.1}$$

where the dependence on the parameters is indicated explicitly on the right-hand side.

What, exactly, do we want to minimize to get fitted values for the a_j's? The first thing that comes to mind is the familiar least-squares fit,

$$\text{minimize over } a_1 \ldots a_M : \qquad \sum_{i=1}^{N} [y_i - y(x_i; a_1 \ldots a_M)]^2 \tag{15.1.2}$$

But where does this come from? What general principles is it based on? The answer to these questions takes us into the subject of *maximum likelihood estimators*.

Given a particular data set of x_i's and y_i's, we have the intuitive feeling that some parameter sets $a_1 \ldots a_M$ are very unlikely — those for which the model function $y(x)$ looks *nothing like* the data — while others may be very likely — those that closely resemble the data. How can we quantify this intuitive feeling? How can we select fitted parameters that are "most likely" to be correct? It is not meaningful to ask the question, "What is the probability that a particular set of fitted parameters $a_1 \ldots a_M$ is correct?" The reason is that there is no statistical universe of models from which the parameters are drawn. There is just one model, the correct one, and a statistical universe of data sets that are drawn from it!

That being the case, we can, however, turn the question around, and ask, "*Given a particular set of parameters*, what is the probability that this data set could have occurred?" If the y_i's take on continuous values, the probability will always be zero unless we add the phrase, "...plus or minus some fixed Δy on each data point." So let's always take this phrase as understood. If the probability of obtaining the data set is infinitesimally small, then we can conclude that the parameters under consideration are "unlikely" to be right. Conversely, our intuition tells us that the data set should not be too improbable for the correct choice of parameters.

In other words, we identify the probability of the data given the parameters (which is a mathematically computable number), as the *likelihood* of the parameters given the data. This identification is entirely based on intuition. It has no formal mathematical basis in and of itself; as we already remarked, statistics is *not* a branch of mathematics!

Once we make this intuitive identification, however, it is only a small further step to decide to fit for the parameters $a_1 \ldots a_M$ precisely by finding those values that *maximize* the likelihood defined in the above way. This form of parameter estimation is *maximum likelihood estimation*.

We are now ready to make the connection to (15.1.2). Suppose that each data point y_i has a measurement error that is independently random and distributed as a normal (Gaussian) distribution around the "true" model $y(x)$. And suppose that the standard deviations σ of these normal distributions are the same for all points. Then the probability of the data set is the product of the probabilities of each point,

$$P \propto \prod_{i=1}^{N} \left\{ \exp\left[-\frac{1}{2} \left(\frac{y_i - y(x_i)}{\sigma} \right)^2 \right] \Delta y \right\} \tag{15.1.3}$$

Notice that there is a factor Δy in each term in the product. Maximizing (15.1.3) is equivalent to maximizing its logarithm, or minimizing the negative of its logarithm, namely,

$$\left[\sum_{i=1}^{N} \frac{[y_i - y(x_i)]^2}{2\sigma^2} \right] - N \log \Delta y \tag{15.1.4}$$

Since N, σ, and Δy are all constants, minimizing this equation is equivalent to minimizing (15.1.2).

What we see is that least-squares fitting *is* a maximum likelihood estimation of the fitted parameters *if* the measurement errors are independent and normally distributed with constant standard deviation. Notice that we made no assumption about the linearity or nonlinearity of the model $y(x; a_1 \ldots)$ in its parameters $a_1 \ldots a_M$. Just below, we will relax our assumption of constant standard deviations and obtain the very similar formulas for what is called "chi-square fitting" or "weighted least-squares fitting." First, however, let us discuss further our very stringent assumption of a normal distribution.

For a hundred years or so, mathematical statisticians have been in love with the fact that the probability distribution of the sum of a very large number of very small random deviations almost always converges to a normal distribution. (For precise statements of this *central limit theorem*, consult [1] or other standard works on mathematical statistics.) This infatuation tended to focus interest away from the

fact that, for real data, the normal distribution is often rather poorly realized, if it is realized at all. We are often taught, rather casually, that, on average, measurements will fall within $\pm\sigma$ of the true value 68 percent of the time, within $\pm 2\sigma$ 95 percent of the time, and within $\pm 3\sigma$ 99.7 percent of the time. Extending this, one would expect a measurement to be off by $\pm 20\sigma$ only one time out of 2×10^{88}. We all know that "glitches" are much more likely than *that*!

In some instances, the deviations from a normal distribution are easy to understand and quantify. For example, in measurements obtained by counting events, the measurement errors are usually distributed as a Poisson distribution, whose cumulative probability function was already discussed in §6.2. When the number of counts going into one data point is large, the Poisson distribution converges towards a Gaussian. However, the convergence is not uniform when measured in fractional accuracy. The more standard deviations out on the tail of the distribution, the larger the number of counts must be before a value close to the Gaussian is realized. The sign of the effect is always the same: The Gaussian predicts that "tail" events are much less likely than they actually (by Poisson) are. This causes such events, when they occur, to skew a least-squares fit much more than they ought.

Other times, the deviations from a normal distribution are not so easy to understand in detail. Experimental points are occasionally just *way off*. Perhaps the power flickered during a point's measurement, or someone kicked the apparatus, or someone wrote down a wrong number. Points like this are called *outliers*. They can easily turn a least-squares fit on otherwise adequate data into nonsense. Their probability of occurrence in the assumed Gaussian model is so small that the maximum likelihood estimator is willing to distort the whole curve to try to bring them, mistakenly, into line.

The subject of *robust statistics* deals with cases where the normal or Gaussian model is a bad approximation, or cases where outliers are important. We will discuss robust methods briefly in §15.7. All the sections between this one and that one assume, one way or the other, a Gaussian model for the measurement errors in the data. It it quite important that you keep the limitations of that model in mind, even as you use the very useful methods that follow from assuming it.

Finally, note that our discussion of measurement errors has been limited to *statistical* errors, the kind that will average away if we only take enough data. Measurements are also susceptible to *systematic* errors that will not go away with any amount of averaging. For example, the calibration of a metal meter stick might depend on its temperature. If we take all our measurements at the same wrong temperature, then no amount of averaging or numerical processing will correct for this unrecognized systematic error.

Chi-Square Fitting

We considered the chi-square statistic once before, in §14.3. Here it arises in a slightly different context.

If each data point (x_i, y_i) has its own, known standard deviation σ_i, then equation (15.1.3) is modified only by putting a subscript i on the symbol σ. That subscript also propagates docilely into (15.1.4), so that the maximum likelihood

estimate of the model parameters is obtained by minimizing the quantity

$$\chi^2 \equiv \sum_{i=1}^{N} \left(\frac{y_i - y(x_i; a_1 \ldots a_M)}{\sigma_i} \right)^2 \tag{15.1.5}$$

called the "chi-square."

To whatever extent the measurement errors actually *are* normally distributed, the quantity χ^2 is correspondingly a sum of N squares of normally distributed quantities, each normalized to unit variance. Once we have adjusted the $a_1 \ldots a_M$ to minimize the value of χ^2, the terms in the sum are not all statistically independent. For models that are linear in the a's, however, it turns out that the probability distribution for different values of χ^2 at its minimum can nevertheless be derived analytically, and is the *chi-square distribution for $N - M$ degrees of freedom*. We learned how to compute this probability function using the incomplete gamma function `gammq` in §6.2. In particular, equation (6.2.18) gives the probability Q that the chi-square should exceed a particular value χ^2 by chance, where $\nu = N - M$ is the *number of degrees of freedom*. The quantity Q, or its complement $P \equiv 1 - Q$, is frequently tabulated in appendices to statistics books, but we generally find it easier to use `gammq` and compute our own values: $Q = \texttt{gammq}(0.5\nu, 0.5\chi^2)$. It is quite common, and usually not too wrong, to assume that the chi-square distribution holds even for models that are not strictly linear in the a's.

This computed probability gives a quantitative measure for the goodness-of-fit of the model. If Q is a very small probability for some particular data set, then the apparent discrepancies are unlikely to be chance fluctuations. Much more probably either (i) the model is wrong — can be statistically rejected, or (ii) someone has lied to you about the size of the measurement errors σ_i — they are really larger than stated.

It is an important point that the chi-square probability Q does not directly measure the credibility of the assumption that the measurement errors are normally distributed. It assumes they are. In most, but not all, cases, however, the effect of nonnormal errors is to create an abundance of outlier points. These decrease the probability Q, so that we can add another possible, though less definitive, conclusion to the above list: (iii) the measurement errors may not be normally distributed.

Possibility (iii) is fairly common, and also fairly benign. It is for this reason that reasonable experimenters are often rather tolerant of low probabilities Q. It is not uncommon to deem acceptable on equal terms any models with, say, $Q > 0.001$. This is not as sloppy as it sounds: Truly *wrong* models will often be rejected with vastly smaller values of Q, 10^{-18}, say. However, if day-in and day-out you find yourself accepting models with $Q \sim 10^{-3}$, you really should track down the cause.

If you happen to know the actual distribution law of your measurement errors, then you might wish to *Monte Carlo simulate* some data sets drawn from a particular model, cf. §7.2–§7.3. You can then subject these synthetic data sets to your actual fitting procedure, so as to determine both the probability distribution of the χ^2 statistic, and also the accuracy with which your model parameters are reproduced by the fit. We discuss this further in §15.6. The technique is very general, but it can also be very expensive.

At the opposite extreme, it sometimes happens that the probability Q is too large, too near to 1, literally too good to be true! Nonnormal measurement errors cannot in general produce this disease, since the normal distribution is about as "compact"

as a distribution can be. Almost always, the cause of too good a chi-square fit is that the experimenter, in a "fit" of conservativism, has *overestimated* his or her measurement errors. Very rarely, too good a chi-square signals actual fraud, data that has been "fudged" to fit the model.

A rule of thumb is that a "typical" value of χ^2 for a "moderately" good fit is $\chi^2 \approx \nu$. More precise is the statement that the χ^2 statistic has a mean ν and a standard deviation $\sqrt{2\nu}$, and, asymptotically for large ν, becomes normally distributed.

In some cases the uncertainties associated with a set of measurements are not known in advance, and considerations related to χ^2 fitting are used to derive a value for σ. If we assume that all measurements have the same standard deviation, $\sigma_i = \sigma$, and that the model does fit well, then we can proceed by first assigning an arbitrary constant σ to all points, next fitting for the model parameters by minimizing χ^2, and finally recomputing

$$\sigma^2 = \sum_{i=1}^{N} [y_i - y(x_i)]^2/(N - M) \tag{15.1.6}$$

Obviously, this approach prohibits an independent assessment of goodness-of-fit, a fact occasionally missed by its adherents. When, however, the measurement error is not known, this approach at least allows *some* kind of error bar to be assigned to the points.

If we take the derivative of equation (15.1.5) with respect to the parameters a_k, we obtain equations that must hold at the chi-square minimum,

$$0 = \sum_{i=1}^{N} \left(\frac{y_i - y(x_i)}{\sigma_i^2} \right) \left(\frac{\partial y(x_i; \ldots a_k \ldots)}{\partial a_k} \right) \qquad k = 1, \ldots, M \tag{15.1.7}$$

Equation (15.1.7) is, in general, a set of M nonlinear equations for the M unknown a_k. Various of the procedures described subsequently in this chapter derive from (15.1.7) and its specializations.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapters 1–4.

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), §VI.C. [1]

15.2 Fitting Data to a Straight Line

A concrete example will make the considerations of the previous section more meaningful. We consider the problem of fitting a set of N data points (x_i, y_i) to a straight-line model

$$y(x) = y(x; a, b) = a + bx \tag{15.2.1}$$

as a distribution can be. Almost always, the cause of too good a chi-square fit is that the experimenter, in a "fit" of conservativism, has *overestimated* his or her measurement errors. Very rarely, too good a chi-square signals actual fraud, data that has been "fudged" to fit the model.

A rule of thumb is that a "typical" value of χ^2 for a "moderately" good fit is $\chi^2 \approx \nu$. More precise is the statement that the χ^2 statistic has a mean ν and a standard deviation $\sqrt{2\nu}$, and, asymptotically for large ν, becomes normally distributed.

In some cases the uncertainties associated with a set of measurements are not known in advance, and considerations related to χ^2 fitting are used to derive a value for σ. If we assume that all measurements have the same standard deviation, $\sigma_i = \sigma$, and that the model does fit well, then we can proceed by first assigning an arbitrary constant σ to all points, next fitting for the model parameters by minimizing χ^2, and finally recomputing

$$\sigma^2 = \sum_{i=1}^{N} [y_i - y(x_i)]^2/(N - M) \tag{15.1.6}$$

Obviously, this approach prohibits an independent assessment of goodness-of-fit, a fact occasionally missed by its adherents. When, however, the measurement error is not known, this approach at least allows *some* kind of error bar to be assigned to the points.

If we take the derivative of equation (15.1.5) with respect to the parameters a_k, we obtain equations that must hold at the chi-square minimum,

$$0 = \sum_{i=1}^{N} \left(\frac{y_i - y(x_i)}{\sigma_i^2} \right) \left(\frac{\partial y(x_i; \ldots a_k \ldots)}{\partial a_k} \right) \qquad k = 1, \ldots, M \tag{15.1.7}$$

Equation (15.1.7) is, in general, a set of M nonlinear equations for the M unknown a_k. Various of the procedures described subsequently in this chapter derive from (15.1.7) and its specializations.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapters 1–4.

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), §VI.C. [1]

15.2 Fitting Data to a Straight Line

A concrete example will make the considerations of the previous section more meaningful. We consider the problem of fitting a set of N data points (x_i, y_i) to a straight-line model

$$y(x) = y(x; a, b) = a + bx \tag{15.2.1}$$

This problem is often called *linear regression*, a terminology that originated, long ago, in the social sciences. We assume that the uncertainty σ_i associated with each measurement y_i is known, and that the x_i's (values of the dependent variable) are known exactly.

To measure how well the model agrees with the data, we use the chi-square merit function (15.1.5), which in this case is

$$\chi^2(a,b) = \sum_{i=1}^{N} \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2 \tag{15.2.2}$$

If the measurement errors are normally distributed, then this merit function will give maximum likelihood parameter estimations of a and b; if the errors are not normally distributed, then the estimations are not maximum likelihood, but may still be useful in a practical sense. In §15.7, we will treat the case where outlier points are so numerous as to render the χ^2 merit function useless.

Equation (15.2.2) is minimized to determine a and b. At its minimum, derivatives of $\chi^2(a,b)$ with respect to a, b vanish.

$$\begin{aligned} 0 &= \frac{\partial \chi^2}{\partial a} = -2 \sum_{i=1}^{N} \frac{y_i - a - bx_i}{\sigma_i^2} \\ 0 &= \frac{\partial \chi^2}{\partial b} = -2 \sum_{i=1}^{N} \frac{x_i (y_i - a - bx_i)}{\sigma_i^2} \end{aligned} \tag{15.2.3}$$

These conditions can be rewritten in a convenient form if we define the following sums:

$$\begin{aligned} S &\equiv \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \quad S_x \equiv \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \quad S_y \equiv \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \\ S_{xx} &\equiv \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} \quad S_{xy} \equiv \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \end{aligned} \tag{15.2.4}$$

With these definitions (15.2.3) becomes

$$\begin{aligned} aS + bS_x &= S_y \\ aS_x + bS_{xx} &= S_{xy} \end{aligned} \tag{15.2.5}$$

The solution of these two equations in two unknowns is calculated as

$$\begin{aligned} \Delta &\equiv SS_{xx} - (S_x)^2 \\ a &= \frac{S_{xx} S_y - S_x S_{xy}}{\Delta} \\ b &= \frac{SS_{xy} - S_x S_y}{\Delta} \end{aligned} \tag{15.2.6}$$

Equation (15.2.6) gives the solution for the best-fit model parameters a and b.

We are not done, however. We must estimate the probable uncertainties in the estimates of a and b, since obviously the measurement errors in the data must introduce some uncertainty in the determination of those parameters. If the data are independent, then each contributes its own bit of uncertainty to the parameters. Consideration of propagation of errors shows that the variance σ_f^2 in the value of any function will be

$$\sigma_f^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial f}{\partial y_i} \right)^2 \tag{15.2.7}$$

For the straight line, the derivatives of a and b with respect to y_i can be directly evaluated from the solution:

$$\begin{aligned} \frac{\partial a}{\partial y_i} &= \frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta} \\ \frac{\partial b}{\partial y_i} &= \frac{S x_i - S_x}{\sigma_i^2 \Delta} \end{aligned} \tag{15.2.8}$$

Summing over the points as in (15.2.7), we get

$$\begin{aligned} \sigma_a^2 &= S_{xx}/\Delta \\ \sigma_b^2 &= S/\Delta \end{aligned} \tag{15.2.9}$$

which are the variances in the estimates of a and b, respectively. We will see in §15.6 that an additional number is also needed to characterize properly the probable uncertainty of the parameter estimation. That number is the *covariance* of a and b, and (as we will see below) is given by

$$\mathrm{Cov}(a, b) = -S_x/\Delta \tag{15.2.10}$$

The coefficient of correlation between the uncertainty in a and the uncertainty in b, which is a number between -1 and 1, follows from (15.2.10) (compare equation 14.5.1),

$$r_{ab} = \frac{-S_x}{\sqrt{S S_{xx}}} \tag{15.2.11}$$

A positive value of r_{ab} indicates that the errors in a and b are likely to have the same sign, while a negative value indicates the errors are anticorrelated, likely to have opposite signs.

We are *still* not done. We must estimate the goodness-of-fit of the data to the model. Absent this estimate, we have not the slightest indication that the parameters a and b in the model have any meaning at all! The probability Q that a value of chi-square as *poor* as the value (15.2.2) should occur by chance is

$$Q = \texttt{gammq}\left(\frac{N-2}{2}, \frac{\chi^2}{2} \right) \tag{15.2.12}$$

Here gammq is our routine for the incomplete gamma function $Q(a,x)$, §6.2. If Q is larger than, say, 0.1, then the goodness-of-fit is believable. If it is larger than, say, 0.001, then the fit *may* be acceptable if the errors are nonnormal or have been moderately underestimated. If Q is less than 0.001 then the model and/or estimation procedure can rightly be called into question. In this latter case, turn to §15.7 to proceed further.

If you do not know the individual measurement errors of the points σ_i, and are proceeding (dangerously) to use equation (15.1.6) for estimating these errors, then here is the procedure for estimating the probable uncertainties of the parameters a and b: Set $\sigma_i \equiv 1$ in all equations through (15.2.6), and multiply σ_a and σ_b, as obtained from equation (15.2.9), by the additional factor $\sqrt{\chi^2/(N-2)}$, where χ^2 is computed by (15.2.2) using the fitted parameters a and b. As discussed above, this procedure is equivalent to *assuming* a good fit, so you get no independent goodness-of-fit probability Q.

In §14.5 we promised a relation between the linear correlation coefficient r (equation 14.5.1) and a goodness-of-fit measure, χ^2 (equation 15.2.2). For unweighted data (all $\sigma_i = 1$), that relation is

$$\chi^2 = (1-r^2)\text{NVar}\,(y_1 \ldots y_N) \tag{15.2.13}$$

where

$$\text{NVar}\,(y_1 \ldots y_N) \equiv \sum_{i=1}^{N} (y_i - \overline{y})^2 \tag{15.2.14}$$

For data with varying weights σ_i, the above equations remain valid if the sums in equation (14.5.1) are weighted by $1/\sigma_i^2$.

The following subroutine, fit, carries out exactly the operations that we have discussed. When the weights σ are known in advance, the calculations exactly correspond to the formulas above. However, when weights σ are unavailable, the routine *assumes* equal values of σ for each point and *assumes* a good fit, as discussed in §15.1.

The formulas (15.2.6) are susceptible to roundoff error. Accordingly, we rewrite them as follows: Define

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right), \qquad i = 1, 2, \ldots, N \tag{15.2.15}$$

and

$$S_{tt} = \sum_{i=1}^{N} t_i^2 \tag{15.2.16}$$

Then, as you can verify by direct substitution,

$$b = \frac{1}{S_{tt}} \sum_{i=1}^{N} \frac{t_i y_i}{\sigma_i} \tag{15.2.17}$$

$$a = \frac{S_y - S_x b}{S} \tag{15.2.18}$$

$$\sigma_a^2 = \frac{1}{S}\left(1 + \frac{S_x^2}{SS_{tt}}\right) \tag{15.2.19}$$

$$\sigma_b^2 = \frac{1}{S_{tt}} \tag{15.2.20}$$

$$\mathrm{Cov}(a,b) = -\frac{S_x}{SS_{tt}} \tag{15.2.21}$$

$$r_{ab} = \frac{\mathrm{Cov}(a,b)}{\sigma_a \sigma_b} \tag{15.2.22}$$

```
      SUBROUTINE fit(x,y,ndata,sig,mwt,a,b,siga,sigb,chi2,q)
      INTEGER mwt,ndata
      REAL a,b,chi2,q,siga,sigb,sig(ndata),x(ndata),y(ndata)
C     USES gammq
```

Given a set of data points `x(1:ndata)`,`y(1:ndata)` with individual standard deviations `sig(1:ndata)`, fit them to a straight line $y = a + bx$ by minimizing χ^2. Returned are `a`,`b` and their respective probable uncertainties `siga` and `sigb`, the chi-square `chi2`, and the goodness-of-fit probability `q` (that the fit would have χ^2 this large or larger). If `mwt=0` on input, then the standard deviations are assumed to be unavailable: `q` is returned as `1.0` and the normalization of `chi2` is to unit standard deviation on all points.

```
      INTEGER i
      REAL sigdat,ss,st2,sx,sxoss,sy,t,wt,gammq
      sx=0.                                   Initialize sums to zero.
      sy=0.
      st2=0.
      b=0.
      if(mwt.ne.0) then                       Accumulate sums ...
        ss=0.
        do 11 i=1,ndata                       ...with weights
          wt=1./(sig(i)**2)
          ss=ss+wt
          sx=sx+x(i)*wt
          sy=sy+y(i)*wt
        enddo 11
      else
        do 12 i=1,ndata                       ...or without weights.
          sx=sx+x(i)
          sy=sy+y(i)
        enddo 12
        ss=float(ndata)
      endif
      sxoss=sx/ss
      if(mwt.ne.0) then
        do 13 i=1,ndata
          t=(x(i)-sxoss)/sig(i)
          st2=st2+t*t
          b=b+t*y(i)/sig(i)
        enddo 13
      else
        do 14 i=1,ndata
          t=x(i)-sxoss
          st2=st2+t*t
          b=b+t*y(i)
        enddo 14
      endif
      b=b/st2                                 Solve for a, b, σa, and σb.
      a=(sy-sx*b)/ss
      siga=sqrt((1.+sx*sx/(ss*st2))/ss)
      sigb=sqrt(1./st2)
      chi2=0.                                 Calculate χ2.
```

```
q=1.
if(mwt.eq.0) then
    do 15 i=1,ndata
        chi2=chi2+(y(i)-a-b*x(i))**2
    enddo 15
    sigdat=sqrt(chi2/(ndata-2))              For unweighted data evaluate typical sig us-
    siga=siga*sigdat                             ing chi2, and adjust the standard devia-
    sigb=sigb*sigdat                             tions.
else
    do 16 i=1,ndata
        chi2=chi2+((y(i)-a-b*x(i))/sig(i))**2
    enddo 16
    if(ndata.gt.2) q=gammq(0.5*(ndata-2),0.5*chi2)      Equation (15.2.12).
endif
return
END
```

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 6.

15.3 Straight-Line Data with Errors in Both Coordinates

If experimental data are subject to measurement error not only in the y_i's, but also in the x_i's, then the task of fitting a straight-line model

$$y(x) = a + bx \tag{15.3.1}$$

is considerably harder. It is straightforward to write down the χ^2 merit function for this case,

$$\chi^2(a,b) = \sum_{i=1}^{N} \frac{(y_i - a - bx_i)^2}{\sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2} \tag{15.3.2}$$

where $\sigma_{x\,i}$ and $\sigma_{y\,i}$ are, respectively, the x and y standard deviations for the ith point. The weighted sum of variances in the denominator of equation (15.3.2) can be understood both as the variance in the direction of the smallest χ^2 between each data point and the line with slope b, and also as the variance of the linear combination $y_i - a - bx_i$ of two random variables x_i and y_i,

$$\mathrm{Var}(y_i - a - bx_i) = \mathrm{Var}(y_i) + b^2\mathrm{Var}(x_i) = \sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2 \equiv 1/w_i \tag{15.3.3}$$

The sum of the square of N random variables, each normalized by its variance, is thus χ^2-distributed.

We want to minimize equation (15.3.2) with respect to a and b. Unfortunately, the occurrence of b in the denominator of equation (15.3.2) makes the resulting equation for the slope $\partial\chi^2/\partial b = 0$ nonlinear. However, the corresponding condition for the intercept, $\partial\chi^2/\partial a = 0$, is still linear and yields

$$a = \left[\sum_i w_i(y_i - bx_i)\right] \Bigg/ \sum_i w_i \tag{15.3.4}$$

where the w_i's are defined by equation (15.3.3). A reasonable strategy, now, is to use the machinery of Chapter 10 (e.g., the routine `brent`) for minimizing a general one-dimensional

```
q=1.
if(mwt.eq.0) then
    do 15 i=1,ndata
        chi2=chi2+(y(i)-a-b*x(i))**2
    enddo 15
    sigdat=sqrt(chi2/(ndata-2))              For unweighted data evaluate typical sig us-
    siga=siga*sigdat                             ing chi2, and adjust the standard devia-
    sigb=sigb*sigdat                             tions.
else
    do 16 i=1,ndata
        chi2=chi2+((y(i)-a-b*x(i))/sig(i))**2
    enddo 16
    if(ndata.gt.2) q=gammq(0.5*(ndata-2),0.5*chi2)      Equation (15.2.12).
endif
return
END
```

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 6.

15.3 Straight-Line Data with Errors in Both Coordinates

If experimental data are subject to measurement error not only in the y_i's, but also in the x_i's, then the task of fitting a straight-line model

$$y(x) = a + bx \tag{15.3.1}$$

is considerably harder. It is straightforward to write down the χ^2 merit function for this case,

$$\chi^2(a,b) = \sum_{i=1}^{N} \frac{(y_i - a - bx_i)^2}{\sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2} \tag{15.3.2}$$

where $\sigma_{x\,i}$ and $\sigma_{y\,i}$ are, respectively, the x and y standard deviations for the ith point. The weighted sum of variances in the denominator of equation (15.3.2) can be understood both as the variance in the direction of the smallest χ^2 between each data point and the line with slope b, and also as the variance of the linear combination $y_i - a - bx_i$ of two random variables x_i and y_i,

$$\mathrm{Var}(y_i - a - bx_i) = \mathrm{Var}(y_i) + b^2\mathrm{Var}(x_i) = \sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2 \equiv 1/w_i \tag{15.3.3}$$

The sum of the square of N random variables, each normalized by its variance, is thus χ^2-distributed.

We want to minimize equation (15.3.2) with respect to a and b. Unfortunately, the occurrence of b in the denominator of equation (15.3.2) makes the resulting equation for the slope $\partial\chi^2/\partial b = 0$ nonlinear. However, the corresponding condition for the intercept, $\partial\chi^2/\partial a = 0$, is still linear and yields

$$a = \left[\sum_i w_i(y_i - bx_i)\right] \Bigg/ \sum_i w_i \tag{15.3.4}$$

where the w_i's are defined by equation (15.3.3). A reasonable strategy, now, is to use the machinery of Chapter 10 (e.g., the routine `brent`) for minimizing a general one-dimensional

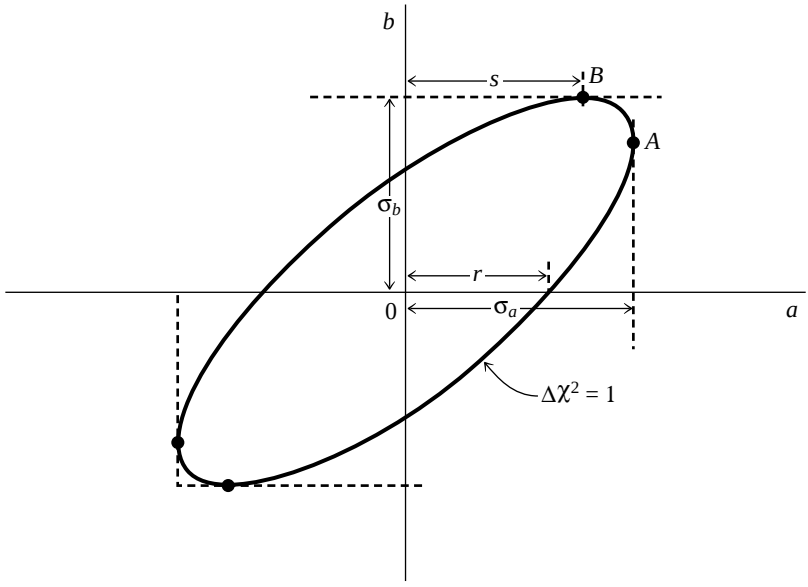


Figure 15.3.1. Standard errors for the parameters a and b. The point B can be found by varying the slope b while simultaneously minimizing the intercept a. This gives the standard error σ_b, and also the value s. The standard error σ_a can then be found by the geometric relation $\sigma_a^2 = s^2 + r^2$.

function to minimize with respect to b, while using equation (15.3.4) at each stage to ensure that the minimum with respect to b is also minimized with respect to a.

Because of the finite error bars on the x_i's, the minimum χ^2 as a function of b will be finite, though usually large, when b equals infinity (line of infinite slope). The angle $\theta \equiv \arctan b$ is thus more suitable as a parametrization of slope than b itself. The value of χ^2 will then be periodic in θ with period π (not 2π!). If any data points have very small σ_y's but moderate or large σ_x's, then it is also possible to have a maximum in χ^2 near zero slope, $\theta \approx 0$. In that case, there can sometimes be two χ^2 minima, one at positive slope and the other at negative. Only one of these is the correct global minimum. It is therefore important to have a good starting guess for b (or θ). Our strategy, implemented below, is to scale the y_i's so as to have variance equal to the x_i's, then to do a conventional (as in §15.2) linear fit with weights derived from the (scaled) sum $\sigma_{y\,i}^2 + \sigma_{x\,i}^2$. This yields a good starting guess for b if the data are even *plausibly* related to a straight-line model.

Finding the standard errors σ_a and σ_b on the parameters a and b is more complicated. We will see in §15.6 that, in appropriate circumstances, the standard errors in a and b are the respective projections onto the a and b axes of the "confidence region boundary" where χ^2 takes on a value one greater than its minimum, $\Delta\chi^2 = 1$. In the linear case of §15.2, these projections follow from the Taylor series expansion

$$\Delta\chi^2 \approx \frac{1}{2}\left[\frac{\partial^2\chi^2}{\partial a^2}(\Delta a)^2 + \frac{\partial^2\chi^2}{\partial b^2}(\Delta b)^2\right] + \frac{\partial^2\chi^2}{\partial a\partial b}\Delta a\Delta b \tag{15.3.5}$$

Because of the present nonlinearity in b, however, analytic formulas for the second derivatives are quite unwieldy; more important, the lowest-order term frequently gives a poor approximation to $\Delta\chi^2$. Our strategy is therefore to find the roots of $\Delta\chi^2 = 1$ numerically, by adjusting the value of the slope b away from the minimum. In the program below the general root finder `zbrent` is used. It may occur that there are no roots at all — for example, if all error bars are so large that all the data points are compatible with each other. It is important, therefore, to make some effort at bracketing a putative root before refining it (cf. §9.1).

Because a is minimized at each stage of varying b, successful numerical root-finding leads to a value of Δa that minimizes χ^2 for the value of Δb that gives $\Delta\chi^2 = 1$. This (see Figure 15.3.1) directly gives the tangent projection of the confidence region onto the b axis,

and thus σ_b. It does not, however, give the tangent projection of the confidence region onto the a axis. In the figure, we have found the point labeled B; to find σ_a we need to find the point A. Geometry to the rescue: To the extent that the confidence region is approximated by an ellipse, then you can prove (see figure) that $\sigma_a^2 = r^2 + s^2$. The value of s is known from having found the point B. The value of r follows from equations (15.3.2) and (15.3.3) applied at the χ^2 minimum (point O in the figure), giving

$$r^2 = 1 \Big/ \sum_i w_i \tag{15.3.6}$$

Actually, since b can go through infinity, this whole procedure makes more sense in (a, θ) space than in (a, b) space. That is in fact how the following program works. Since it is conventional, however, to return standard errors for a and b, not a and θ, we finally use the relation

$$\sigma_b = \sigma_\theta / \cos^2 \theta \tag{15.3.7}$$

We caution that if b and its standard error are both large, so that the confidence region actually includes infinite slope, then the standard error σ_b is not very meaningful. The function `chixy` is normally called only by the routine `fitexy`. However, if you want, you can yourself explore the confidence region by making repeated calls to `chixy` (whose argument is an angle θ, not a slope b), after a single initializing call to `fitexy`.

A final caution, repeated from §15.0, is that if the goodness-of-fit is not acceptable (returned probability is too small), the standard errors σ_a and σ_b are surely not believable. In dire circumstances, you might try scaling all your x and y error bars by a constant factor until the probability is acceptable (0.5, say), to get more plausible values for σ_a and σ_b.

```
      SUBROUTINE fitexy(x,y,ndat,sigx,sigy,a,b,siga,sigb,chi2,q)
      INTEGER ndat,NMAX
      REAL x(ndat),y(ndat),sigx(ndat),sigy(ndat),a,b,siga,sigb,chi2,
     *     q,POTN,PI,BIG,ACC
      PARAMETER (NMAX=1000,POTN=1.571000,BIG=1.e30,PI=3.14159265,
     *     ACC=1.e-3)
C     USES avevar,brent,chixy,fit,gammq,mnbrak,zbrent
```

Straight-line fit to input data `x(1:ndat)` and `y(1:ndat)` with errors in both x and y, the respective standard deviations being the input quantities `sigx(1:ndat)` and `sigy(1:ndat)`. Output quantities are `a` and `b` such that $y = a + bx$ minimizes χ^2, whose value is returned as `chi2`. The χ^2 probability is returned as `q`, a small value indicating a poor fit (sometimes indicating underestimated errors). Standard errors on `a` and `b` are returned as `siga` and `sigb`. These are not meaningful if either (i) the fit is poor, or (ii) b is so large that the data are consistent with a vertical (infinite b) line. If `siga` and `sigb` are returned as `BIG`, then the data are consistent with *all* values of b.

```
      INTEGER j,nn
      REAL xx(NMAX),yy(NMAX),sx(NMAX),sy(NMAX),ww(NMAX),swap,amx,amn
     *     ,varx,vary,aa,offs,ang(6),ch(6),scale,bmn,bmx,d1,d2
     *     ,r2,dum1,dum2,dum3,dum4,dum5,brent,chixy,gammq,zbrent
      COMMON /fitxyc/ xx,yy,sx,sy,ww,aa,offs,nn
      EXTERNAL chixy
      if (ndat.gt.NMAX) pause 'NMAX too small in fitexy'
      call avevar(x,ndat,dum1,varx)          Find the x and y variances, and scale the data
      call avevar(y,ndat,dum1,vary)              into the common block for communication
      scale=sqrt(varx/vary)                      with the function chixy.
      nn=ndat
      do 11 j=1,ndat
        xx(j)=x(j)
        yy(j)=y(j)*scale
        sx(j)=sigx(j)
        sy(j)=sigy(j)*scale
        ww(j)=sqrt(sx(j)**2+sy(j)**2)        Use both x and y weights in first trial fit.
      enddo 11
      call fit(xx,yy,nn,ww,1,dum1,b,dum2,dum3,dum4,dum5)        Trial fit for b.
      offs=0.
```

```
      ang(1)=0.                                  Construct several angles for reference points.
      ang(2)=atan(b)                             Make b an angle.
      ang(4)=0.
      ang(5)=ang(2)
      ang(6)=POTN
      do 12 j=4,6
        ch(j)=chixy(ang(j))
      enddo 12
      call mnbrak(ang(1),ang(2),ang(3),ch(1),ch(2),ch(3),chixy)      Bracket the χ² min-
      chi2=brent(ang(1),ang(2),ang(3),chixy,ACC,b)      imum and then locate it with brent.
      chi2=chixy(b)
      a=aa
      q=gammq(0.5*(nn-2),0.5*chi2)               Compute χ² probability.
      r2=0.
      do 13 j=1,nn                               Save the inverse sum of weights at the mini-
        r2=r2+ww(j)                                  mum.
      enddo 13
      r2=1./r2
      bmx=BIG                                    Now, find standard errors for b as points where
      bmn=BIG                                        Δχ² = 1.
      offs=chi2+1.
      do 14 j=1,6                                Go through saved values to bracket the desired
        if (ch(j).gt.offs) then                      roots. Note periodicity in slope angles.
          d1=mod(abs(ang(j)-b),PI)
          d2=PI-d1
          if(ang(j).lt.b)then
            swap=d1
            d1=d2
            d2=swap
          endif
          if (d1.lt.bmx) bmx=d1
          if (d2.lt.bmn) bmn=d2
        endif
      enddo 14
      if (bmx.lt. BIG) then                      Call zbrent to find the roots.
        bmx=zbrent(chixy,b,b+bmx,ACC)-b
        amx=aa-a
        bmn=zbrent(chixy,b,b-bmn,ACC)-b
        amn=aa-a
        sigb=sqrt(0.5*(bmx**2+bmn**2))/(scale*cos(b)**2)
        siga=sqrt(0.5*(amx**2+amn**2)+r2)/scale      Error in a has additional piece r2.
      else
        sigb=BIG
        siga=BIG
      endif
      a=a/scale                                  Unscale the answers.
      b=tan(b)/scale
      return
      END



      FUNCTION chixy(bang)
      REAL chixy,bang,BIG
      INTEGER NMAX
      PARAMETER (NMAX=1000,BIG=1.E30)
        Captive function of fitexy, returns the value of (χ² − offs) for the slope b=tan(bang).
        Scaled data and offs are communicated via the common block /fitxyc/.
      INTEGER nn,j
      REAL xx(NMAX),yy(NMAX),sx(NMAX),sy(NMAX),ww(NMAX),aa,offs,
     *     avex,avey,sumw,b
      COMMON /fitxyc/ xx,yy,sx,sy,ww,aa,offs,nn
      b=tan(bang)
      avex=0.
```

```
avey=0.
sumw=0.
do 11 j=1,nn
    ww(j)=(b*sx(j))**2+sy(j)**2
    if(ww(j).lt.1./BIG) then
        ww(j)=BIG
    else
        ww(j)=1./ww(j)
    endif
    sumw=sumw+ww(j)
    avex=avex+ww(j)*xx(j)
    avey=avey+ww(j)*yy(j)
enddo 11
avex=avex/sumw
avey=avey/sumw
aa=avey-b*avex
chixy=-offs
do 12 j=1,nn
    chixy=chixy+ww(j)*(yy(j)-aa-b*xx(j))**2
enddo 12
return
END
```

Be aware that the literature on the seemingly straightforward subject of this section is generally confusing and sometimes plain wrong. Deming's [1] early treatment is sound, but its reliance on Taylor expansions gives inaccurate error estimates. References [2-4] are reliable, more recent, general treatments with critiques of earlier work. York [5] and Reed [6] usefully discuss the simple case of a straight line as treated here, but the latter paper has some errors, corrected in [7]. All this commotion has attracted the Bayesians [8-10], who have still different points of view.

CITED REFERENCES AND FURTHER READING:

Deming, W.E. 1943, *Statistical Adjustment of Data* (New York: Wiley), reprinted 1964 (New York: Dover). [1]

Jefferys, W.H. 1980, *Astronomical Journal*, vol. 85, pp. 177–181; see also vol. 95, p. 1299 (1988). [2]

Jefferys, W.H. 1981, *Astronomical Journal*, vol. 86, pp. 149–155; see also vol. 95, p. 1300 (1988). [3]

Lybanon, M. 1984, *American Journal of Physics*, vol. 52, pp. 22–26. [4]

York, D. 1966, *Canadian Journal of Physics*, vol. 44, pp. 1079–1086. [5]

Reed, B.C. 1989, *American Journal of Physics*, vol. 57, pp. 642–646; see also vol. 58, p. 189, and vol. 58, p. 1209. [6]

Reed, B.C. 1992, *American Journal of Physics*, vol. 60, pp. 59–62. [7]

Zellner, A. 1971, *An Introduction to Bayesian Inference in Econometrics* (New York: Wiley); reprinted 1987 (Malabar, FL: R. E. Krieger Pub. Co.). [8]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [9]

Jaynes, E.T. 1991, in *Maximum-Entropy and Bayesian Methods, Proc. 10th Int. Workshop*, W.T. Grandy, Jr., and L.H. Schick, eds. (Boston: Kluwer). [10]

Macdonald, J.R., and Thompson, W.J. 1992, *American Journal of Physics*, vol. 60, pp. 66–73.

15.4 General Linear Least Squares

An immediate generalization of §15.2 is to fit a set of data points (x_i, y_i) to a model that is not just a linear combination of 1 and x (namely $a + bx$), but rather a linear combination of *any* M specified functions of x. For example, the functions could be $1, x, x^2, \ldots, x^{M-1}$, in which case their general linear combination,

$$y(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_M x^{M-1} \tag{15.4.1}$$

is a polynomial of degree $M - 1$. Or, the functions could be sines and cosines, in which case their general linear combination is a harmonic series.

The general form of this kind of model is

$$y(x) = \sum_{k=1}^{M} a_k X_k(x) \tag{15.4.2}$$

where $X_1(x), \ldots, X_M(x)$ are arbitrary fixed functions of x, called the *basis functions*.

Note that the functions $X_k(x)$ can be wildly nonlinear functions of x. In this discussion "linear" refers only to the model's dependence on its *parameters* a_k.

For these linear models we generalize the discussion of the previous section by defining a merit function

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{15.4.3}$$

As before, σ_i is the measurement error (standard deviation) of the ith data point, presumed to be known. If the measurement errors are not known, they may all (as discussed at the end of §15.1) be set to the constant value $\sigma = 1$.

Once again, we will pick as best parameters those that minimize χ^2. There are several different techniques available for finding this minimum. Two are particularly useful, and we will discuss both in this section. To introduce them and elucidate their relationship, we need some notation.

Let $\mathbf{A}$ be a matrix whose $N \times M$ components are constructed from the M basis functions evaluated at the N abscissas x_i, and from the N measurement errors σ_i, by the prescription

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} \tag{15.4.4}$$

The matrix $\mathbf{A}$ is called the *design matrix* of the fitting problem. Notice that in general $\mathbf{A}$ has more rows than columns, $N \geq M$, since there must be more data points than model parameters to be solved for. (You can fit a straight line to two points, but not a very meaningful quintic!) The design matrix is shown schematically in Figure 15.4.1.

Also define a vector $\mathbf{b}$ of length N by

$$b_i = \frac{y_i}{\sigma_i} \tag{15.4.5}$$

and denote the M vector whose components are the parameters to be fitted, $a_1, \ldots, a_M$, by $\mathbf{a}$.

15.4 General Linear Least Squares

An immediate generalization of §15.2 is to fit a set of data points (x_i, y_i) to a model that is not just a linear combination of 1 and x (namely $a + bx$), but rather a linear combination of *any* M specified functions of x. For example, the functions could be $1, x, x^2, \ldots, x^{M-1}$, in which case their general linear combination,

$$y(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_M x^{M-1} \tag{15.4.1}$$

is a polynomial of degree $M - 1$. Or, the functions could be sines and cosines, in which case their general linear combination is a harmonic series.

The general form of this kind of model is

$$y(x) = \sum_{k=1}^{M} a_k X_k(x) \tag{15.4.2}$$

where $X_1(x), \ldots, X_M(x)$ are arbitrary fixed functions of x, called the *basis functions*.

Note that the functions $X_k(x)$ can be wildly nonlinear functions of x. In this discussion "linear" refers only to the model's dependence on its *parameters* a_k.

For these linear models we generalize the discussion of the previous section by defining a merit function

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{15.4.3}$$

As before, σ_i is the measurement error (standard deviation) of the ith data point, presumed to be known. If the measurement errors are not known, they may all (as discussed at the end of §15.1) be set to the constant value $\sigma = 1$.

Once again, we will pick as best parameters those that minimize χ^2. There are several different techniques available for finding this minimum. Two are particularly useful, and we will discuss both in this section. To introduce them and elucidate their relationship, we need some notation.

Let $\mathbf{A}$ be a matrix whose $N \times M$ components are constructed from the M basis functions evaluated at the N abscissas x_i, and from the N measurement errors σ_i, by the prescription

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} \tag{15.4.4}$$

The matrix $\mathbf{A}$ is called the *design matrix* of the fitting problem. Notice that in general $\mathbf{A}$ has more rows than columns, $N \geq M$, since there must be more data points than model parameters to be solved for. (You can fit a straight line to two points, but not a very meaningful quintic!) The design matrix is shown schematically in Figure 15.4.1.

Also define a vector $\mathbf{b}$ of length N by

$$b_i = \frac{y_i}{\sigma_i} \tag{15.4.5}$$

and denote the M vector whose components are the parameters to be fitted, $a_1, \ldots, a_M$, by $\mathbf{a}$.

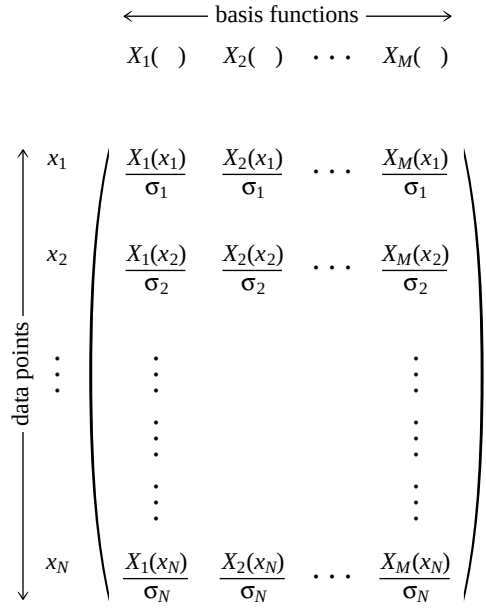


Figure 15.4.1. Design matrix for the least-squares fit of a linear combination of M basis functions to N data points. The matrix elements involve the basis functions evaluated at the values of the independent variable at which measurements are made, and the standard deviations of the measured dependent variable. The measured values of the dependent variable do not enter the design matrix.

Solution by Use of the Normal Equations

The minimum of (15.4.3) occurs where the derivative of χ^2 with respect to all M parameters a_k vanishes. Specializing equation (15.1.7) to the case of the model (15.4.2), this condition yields the M equations

$$0 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=1}^{M} a_j X_j(x_i) \right] X_k(x_i) \qquad k = 1, \ldots, M \tag{15.4.6}$$

Interchanging the order of summations, we can write (15.4.6) as the matrix equation

$$\sum_{j=1}^{M} \alpha_{kj} a_j = \beta_k \tag{15.4.7}$$

where

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2} \qquad \text{or equivalently} \qquad [\alpha] = \mathbf{A}^T \cdot \mathbf{A} \tag{15.4.8}$$

an $M \times M$ matrix, and

$$\beta_k = \sum_{i=1}^{N} \frac{y_i X_k(x_i)}{\sigma_i^2} \qquad \text{or equivalently} \qquad [\beta] = \mathbf{A}^T \cdot \mathbf{b} \tag{15.4.9}$$

a vector of length M.

The equations (15.4.6) or (15.4.7) are called the *normal equations* of the least-squares problem. They can be solved for the vector of parameters **a** by the standard methods of Chapter 2, notably LU decomposition and backsubstitution, Choleksy decomposition, or Gauss-Jordan elimination. In matrix form, the normal equations can be written as either

$$[\alpha] \cdot \mathbf{a} = [\beta] \qquad \text{or as} \qquad \left(\mathbf{A}^T \cdot \mathbf{A}\right) \cdot \mathbf{a} = \mathbf{A}^T \cdot \mathbf{b} \tag{15.4.10}$$

The inverse matrix $C_{jk} \equiv [\alpha]^{-1}_{jk}$ is closely related to the probable (or, more precisely, *standard*) uncertainties of the estimated parameters **a**. To estimate these uncertainties, consider that

$$a_j = \sum_{k=1}^{M} [\alpha]^{-1}_{jk} \beta_k = \sum_{k=1}^{M} C_{jk} \left[\sum_{i=1}^{N} \frac{y_i X_k(x_i)}{\sigma_i^2} \right] \tag{15.4.11}$$

and that the variance associated with the estimate a_j can be found as in (15.2.7) from

$$\sigma^2(a_j) = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial a_j}{\partial y_i} \right)^2 \tag{15.4.12}$$

Note that α_{jk} is independent of y_i, so that

$$\frac{\partial a_j}{\partial y_i} = \sum_{k=1}^{M} C_{jk} X_k(x_i)/\sigma_i^2 \tag{15.4.13}$$

Consequently, we find that

$$\sigma^2(a_j) = \sum_{k=1}^{M} \sum_{l=1}^{M} C_{jk} C_{jl} \left[\sum_{i=1}^{N} \frac{X_k(x_i) X_l(x_i)}{\sigma_i^2} \right] \tag{15.4.14}$$

The final term in brackets is just the matrix $[\alpha]$. Since this is the matrix inverse of $[C]$, (15.4.14) reduces immediately to

$$\sigma^2(a_j) = C_{jj} \tag{15.4.15}$$

In other words, the diagonal elements of $[C]$ are the variances (squared uncertainties) of the fitted parameters **a**. It should not surprise you to learn that the off-diagonal elements C_{jk} are the covariances between a_j and a_k (cf. 15.2.10); but we shall defer discussion of these to §15.6.

We will now give a routine that implements the above formulas for the general linear least-squares problem, by the method of normal equations. Since we wish to compute not only the solution vector **a** but also the covariance matrix $[C]$, it is most convenient to use Gauss-Jordan elimination (routine `gaussj` of §2.1) to perform the linear algebra. The operation count, in this application, is no larger than that for LU decomposition. If you have no need for the covariance matrix, however, you can save a factor of 3 on the linear algebra by switching to LU decomposition, without

computation of the matrix inverse. In theory, since $\mathbf{A}^T \cdot \mathbf{A}$ is positive definite, Cholesky decomposition is the most efficient way to solve the normal equations. However, in practice most of the computing time is spent in looping over the data to form the equations, and Gauss-Jordan is quite adequate.

We need to warn you that the solution of a least-squares problem directly from the normal equations is rather susceptible to roundoff error. An alternative, and preferred, technique involves QR decomposition (§2.10, §11.3, and §11.6) of the design matrix $\mathbf{A}$. This is essentially what we did at the end of §15.2 for fitting data to a straight line, but without invoking all the machinery of QR to derive the necessary formulas. Later in this section, we will discuss other difficulties in the least-squares problem, for which the cure is *singular value decomposition* (SVD), of which we give an implementation. It turns out that SVD also fixes the roundoff problem, so it is our recommended technique for all but "easy" least-squares problems. It is for these easy problems that the following routine, which solves the normal equations, is intended.

The routine below introduces one bookkeeping trick that is quite useful in practical work. Frequently it is a matter of "art" to decide which parameters a_k in a model should be fit from the data set, and which should be held constant at fixed values, for example values predicted by a theory or measured in a previous experiment. One wants, therefore, to have a convenient means for "freezing" and "unfreezing" the parameters a_k. In the following routine the total number of parameters a_k is denoted `ma` (called M above). As input to the routine, you supply an array `ia(1:ma)`, whose components are either zero or nonzero (e.g., 1). Zeros indicate that you want the corresponding elements of the parameter vector `a(1:ma)` to be held fixed at their input values. Nonzeros indicate parameters that should be fitted for. On output, any frozen parameters will have their variances, and all their covariances, set to zero in the covariance matrix.

```
      SUBROUTINE lfit(x,y,sig,ndat,a,ia,ma,covar,npc,chisq,funcs)
      INTEGER ma,ia(ma),npc,ndat,MMAX
      REAL chisq,a(ma),covar(npc,npc),sig(ndat),x(ndat),y(ndat)
      EXTERNAL funcs
      PARAMETER (MMAX=50)               Set to the maximum number of coefficients ma.
C     USES covsrt,gaussj
        Given a set of data points x(1:ndat), y(1:ndat) with individual standard deviations
        sig(1:ndat), use χ² minimization to fit for some or all of the coefficients a(1:ma) of a
        function that depends linearly on a, y = Σ_i a_i × afunc_i(x). The input array ia(1:ma) in-
        dicates by nonzero entries those components of a that should be fitted for, and by zero entries
        those components that should be held fixed at their input values. The program returns values
        for a(1:ma), χ² = chisq, and the covariance matrix covar(1:ma,1:ma). (Parameters
        held fixed will return zero covariances.) npc is the physical dimension of covar(npc,npc)
        in the calling routine. The user supplies a subroutine funcs(x,afunc,ma) that returns
        the ma basis functions evaluated at x = x in the array afunc.
      INTEGER i,j,k,l,m,mfit
      REAL sig2i,sum,wt,ym,afunc(MMAX),beta(MMAX)
      mfit=0
      do 11 j=1,ma
        if(ia(j).ne.0) mfit=mfit+1
      enddo 11
      if(mfit.eq.0) pause 'lfit: no parameters to be fitted'
      do 13 j=1,mfit                    Initialize the (symmetric) matrix.
        do 12 k=1,mfit
          covar(j,k)=0.
        enddo 12
        beta(j)=0.
      enddo 13
```

```
do 17 i=1,ndat                      Loop over data to accumulate coefficients of the normal
    call funcs(x(i),afunc,ma)           equations.
    ym=y(i)
    if(mfit.lt.ma) then             Subtract off dependences on known pieces of the fitting
        do 14 j=1,ma                    function.
            if(ia(j).eq.0) ym=ym-a(j)*afunc(j)
        enddo 14
    endif
    sig2i=1./sig(i)**2
    j=0
    do 16 l=1,ma
        if (ia(l).ne.0) then
            j=j+1
            wt=afunc(l)*sig2i
            k=0
            do 15 m=1,l
                if (ia(m).ne.0) then
                    k=k+1
                    covar(j,k)=covar(j,k)+wt*afunc(m)
                endif
            enddo 15
            beta(j)=beta(j)+ym*wt
        endif
    enddo 16
enddo 17
do 19 j=2,mfit                      Fill in above the diagonal from symmetry.
    do 18 k=1,j-1
        covar(k,j)=covar(j,k)
    enddo 18
enddo 19
call gaussj(covar,mfit,npc,beta,1,1)   Matrix solution.
j=0
do 21 l=1,ma
    if(ia(l).ne.0) then
        j=j+1
        a(l)=beta(j)                Partition solution to appropriate coefficients a.
    endif
enddo 21
chisq=0.                            Evaluate \chi^2 of the fit.
do 23 i=1,ndat
    call funcs(x(i),afunc,ma)
    sum=0.
    do 22 j=1,ma
        sum=sum+a(j)*afunc(j)
    enddo 22
    chisq=chisq+((y(i)-sum)/sig(i))**2
enddo 23
call covsrt(covar,npc,ma,ia,mfit)      Sort covariance matrix to true order of fitting
return                                     coefficients.
END
```

That last call to a subroutine `covsrt` is only for the purpose of spreading the covariances back into the full `ma` $\times$ `ma` covariance matrix, in the proper rows and columns and with zero variances and covariances set for variables which were held frozen.

The subroutine `covsrt` is as follows.

```
SUBROUTINE covsrt(covar,npc,ma,ia,mfit)
INTEGER ma,mfit,npc,ia(ma)
REAL covar(npc,npc)
    Expand in storage the covariance matrix covar, so as to take into account parameters that
    are being held fixed. (For the latter, return zero covariances.)
INTEGER i,j,k
REAL swap
```

```
do 12 i=mfit+1,ma
    do 11 j=1,i
        covar(i,j)=0.
        covar(j,i)=0.
    enddo 11
enddo 12
k=mfit
do 15 j=ma,1,-1
    if(ia(j).ne.0)then
        do 13 i=1,ma
            swap=covar(i,k)
            covar(i,k)=covar(i,j)
            covar(i,j)=swap
        enddo 13
        do 14 i=1,ma
            swap=covar(k,i)
            covar(k,i)=covar(j,i)
            covar(j,i)=swap
        enddo 14
        k=k-1
    endif
enddo 15
return
END
```

Solution by Use of Singular Value Decomposition

In some applications, the normal equations are perfectly adequate for linear least-squares problems. However, in many cases the normal equations are very close to singular. A zero pivot element may be encountered during the solution of the linear equations (e.g., in `gaussj`), in which case you get no solution at all. Or a very small pivot may occur, in which case you typically get fitted parameters a_k with very large magnitudes that are delicately (and unstably) balanced to cancel out almost precisely when the fitted function is evaluated.

Why does this commonly occur? The reason is that, more often than experimenters would like to admit, data do not clearly distinguish between two or more of the basis functions provided. If two such functions, or two different combinations of functions, happen to fit the data about equally well — or equally badly — then the matrix $[\alpha]$, unable to distinguish between them, neatly folds up its tent and becomes singular. There is a certain mathematical irony in the fact that least-squares problems are *both* overdetermined (number of data points greater than number of parameters) *and* underdetermined (ambiguous combinations of parameters exist); but that is how it frequently is. The ambiguities can be extremely hard to notice *a priori* in complicated problems.

Enter singular value decomposition (SVD). This would be a good time for you to review the material in §2.6, which we will not repeat here. In the case of an overdetermined system, SVD produces a solution that is the best approximation in the least-squares sense, cf. equation (2.6.10). That is exactly what we want. In the case of an underdetermined system, SVD produces a solution whose values (for us, the a_k's) are smallest in the least-squares sense, cf. equation (2.6.8). That is also what we want: When some combination of basis functions is irrelevant to the fit, that combination will be driven down to a small, innocuous, value, rather than pushed up to delicately canceling infinities.

In terms of the design matrix **A** (equation 15.4.4) and the vector **b** (equation 15.4.5), minimization of χ^2 in (15.4.3) can be written as

$$\text{find} \quad \mathbf{a} \quad \text{that minimizes} \quad \chi^2 = |\mathbf{A} \cdot \mathbf{a} - \mathbf{b}|^2 \tag{15.4.16}$$

Comparing to equation (2.6.9), we see that this is precisely the problem that routines `svdcmp` and `svbksb` are designed to solve. The solution, which is given by equation (2.6.12), can be rewritten as follows: If **U** and **V** enter the SVD decomposition of **A** according to equation (2.6.1), as computed by `svdcmp`, then let the vectors $\mathbf{U}_{(i)}\ i = 1, \ldots, M$ denote the *columns* of **U** (each one a vector of length N); and let the vectors $\mathbf{V}_{(i)}; i = 1, \ldots, M$ denote the *columns* of **V** (each one a vector of length M). Then the solution (2.6.12) of the least-squares problem (15.4.16) can be written as

$$\mathbf{a} = \sum_{i=1}^{M} \left(\frac{\mathbf{U}_{(i)} \cdot \mathbf{b}}{w_i} \right) \mathbf{V}_{(i)} \tag{15.4.17}$$

where the w_i are, as in §2.6, the singular values returned by `svdcmp`.

Equation (15.4.17) says that the fitted parameters **a** are linear combinations of the columns of **V**, with coefficients obtained by forming dot products of the columns of **U** with the weighted data vector (15.4.5). Though it is beyond our scope to prove here, it turns out that the standard (loosely, "probable") errors in the fitted parameters are also linear combinations of the columns of **V**. In fact, equation (15.4.17) can be written in a form displaying these errors as

$$\mathbf{a} = \left[\sum_{i=1}^{M} \left(\frac{\mathbf{U}_{(i)} \cdot \mathbf{b}}{w_i} \right) \mathbf{V}_{(i)} \right] \pm \frac{1}{w_1} \mathbf{V}_{(1)} \pm \cdots \pm \frac{1}{w_M} \mathbf{V}_{(M)} \tag{15.4.18}$$

Here each $\pm$ is followed by a standard deviation. The amazing fact is that, decomposed in this fashion, the standard deviations are all mutually independent (uncorrelated). Therefore they can be added together in root-mean-square fashion. What is going on is that the vectors $\mathbf{V}_{(i)}$ are the principal axes of the error ellipsoid of the fitted parameters **a** (see §15.6).

It follows that the variance in the estimate of a parameter a_j is given by

$$\sigma^2(a_j) = \sum_{i=1}^{M} \frac{1}{w_i^2} [\mathbf{V}_{(i)}]_j^2 = \sum_{i=1}^{M} \left(\frac{V_{ji}}{w_i} \right)^2 \tag{15.4.19}$$

whose result should be identical with (15.4.14). As before, you should not be surprised at the formula for the covariances, here given without proof,

$$\text{Cov}(a_j, a_k) = \sum_{i=1}^{M} \left(\frac{V_{ji} V_{ki}}{w_i^2} \right) \tag{15.4.20}$$

We introduced this subsection by noting that the normal equations can fail by encountering a zero pivot. We have not yet, however, mentioned how SVD overcomes this problem. The answer is: If any singular value w_i is zero, its

reciprocal in equation (15.4.18) should be set to zero, not infinity. (Compare the discussion preceding equation 2.6.7.) This corresponds to adding to the fitted parameters **a** a *zero* multiple, rather than some random large multiple, of any linear combination of basis functions that are degenerate in the fit. It is a good thing to do!

Moreover, if a singular value w_i is nonzero but very small, you should also define *its* reciprocal to be zero, since its apparent value is probably an artifact of roundoff error, not a meaningful number. A plausible answer to the question "how small is small?" is to edit in this fashion all singular values whose ratio to the largest singular value is less than N times the machine precision ϵ. (You might argue for $\sqrt{N}$, or a constant, instead of N as the multiple; that starts getting into hardware-dependent questions.)

There is another reason for editing even *additional* singular values, ones large enough that roundoff error is not a question. Singular value decomposition allows you to identify linear combinations of variables that just happen not to contribute much to reducing the χ^2 of your data set. Editing these can sometimes reduce the probable error on your coefficients quite significantly, while increasing the minimum χ^2 only negligibly. We will learn more about identifying and treating such cases in §15.6. In the following routine, the point at which this kind of editing would occur is indicated.

Generally speaking, we recommend that you always use SVD techniques instead of using the normal equations. SVD's only significant disadvantage is that it requires an extra array of size $N \times M$ to store the whole design matrix. This storage is overwritten by the matrix **U**. Storage is also required for the $M \times M$ matrix **V**, but this is instead of the same-sized matrix for the coefficients of the normal equations. SVD can be significantly slower than solving the normal equations; however, its great advantage, that it (theoretically) *cannot fail*, more than makes up for the speed disadvantage.

In the routine that follows, the matrices `u`, `v` and the vector `w` are input as working space. `np` and `mp` are their various physical dimensions. The logical dimensions of the problem are `ndata` data points by `ma` basis functions (and fitted parameters). If you care only about the values `a` of the fitted parameters, then `u`, `v`, `w` contain no useful information on output. If you want probable errors for the fitted parameters, read on.

```
      SUBROUTINE svdfit(x,y,sig,ndata,a,ma,u,v,w,mp,np,
     *     chisq,funcs)
      INTEGER ma,mp,ndata,np,NMAX,MMAX
      REAL chisq,a(ma),sig(ndata),u(mp,np),v(np,np),w(np),
     *     x(ndata),y(ndata),TOL
      EXTERNAL funcs
      PARAMETER (NMAX=1000,MMAX=50,TOL=1.e-5)  Max expected ndata and ma.
C     USES svbksb,svdcmp
```

Given a set of data points `x(1:ndata)`,`y(1:ndata)` with individual standard deviations `sig(1:ndata)`, use χ^2 minimization to determine the `ma` coefficients `a` of the fitting function $y = \sum_i \mathtt{a}_i \times \mathtt{afunc}_i(x)$. Here we solve the fitting equations using singular value decomposition of the `ndata` by `ma` matrix, as in §2.6. Arrays `u(1:mp,1:np)`, `v(1:np,1:np)`, `w(1:np)` provide workspace on input; on output they define the singular value decomposition, and can be used to obtain the covariance matrix. `mp`,`np` are the physical dimensions of the matrices `u`,`v`,`w`, as indicated above. It is necessary that `mp`≥`ndata`, `np` ≥ `ma`. The program returns values for the `ma` fit parameters `a`, and χ^2, `chisq`. The user supplies a subroutine `funcs(x,afunc,ma)` that returns the `ma` basis functions evaluated at $x =$ `x` in the array `afunc`.

```
      INTEGER i,j
      REAL sum,thresh,tmp,wmax,afunc(MMAX),b(NMAX)
```

```
do 12 i=1,ndata                                  Accumulate coefficients of the fitting ma-
    call funcs(x(i),afunc,ma)                        trix.
    tmp=1./sig(i)
    do 11 j=1,ma
        u(i,j)=afunc(j)*tmp
    enddo 11
    b(i)=y(i)*tmp
enddo 12
call svdcmp(u,ndata,ma,mp,np,w,v)                Singular value decomposition.
wmax=0.                                          Edit the singular values, given TOL from the
do 13 j=1,ma                                         parameter statement, between here ...
    if(w(j).gt.wmax)wmax=w(j)
enddo 13
thresh=TOL*wmax
do 14 j=1,ma
    if(w(j).lt.thresh)w(j)=0.
enddo 14                                         ...and here.
call svbksb(u,w,v,ndata,ma,mp,np,b,a)
chisq=0.                                         Evaluate chi-square.
do 16 i=1,ndata
    call funcs(x(i),afunc,ma)
    sum=0.
    do 15 j=1,ma
        sum=sum+a(j)*afunc(j)
    enddo 15
    chisq=chisq+((y(i)-sum)/sig(i))**2
enddo 16
return
END
```

Feeding the matrix `v` and vector `w` output by the above program into the following short routine, you easily obtain variances and covariances of the fitted parameters `a`. The square roots of the variances are the standard deviations of the fitted parameters. The routine straightforwardly implements equation (15.4.20) above, with the convention that singular values equal to zero are recognized as having been edited out of the fit.

```
SUBROUTINE svdvar(v,ma,np,w,cvm,ncvm)
INTEGER ma,ncvm,np,MMAX
REAL cvm(ncvm,ncvm),v(np,np),w(np)
PARAMETER (MMAX=20)              Set to the maximum number of fit parameters.
    To evaluate the covariance matrix cvm of the fit for ma parameters obtained by svdfit,
    call this routine with matrices v,w as returned from svdfit. np,ncvm give the physical
    dimensions of v,w,cvm as indicated.
INTEGER i,j,k
REAL sum,wti(MMAX)
do 11 i=1,ma
    wti(i)=0.
    if(w(i).ne.0.) wti(i)=1./(w(i)*w(i))
enddo 11
do 14 i=1,ma                     Sum contributions to covariance matrix (15.4.20).
    do 13 j=1,i
        sum=0.
        do 12 k=1,ma
            sum=sum+v(i,k)*v(j,k)*wti(k)
        enddo 12
        cvm(i,j)=sum
        cvm(j,i)=sum
    enddo 13
enddo 14
return
END
```

Examples

Be aware that some apparently nonlinear problems can be expressed so that they are linear. For example, an exponential model with two parameters a and b,

$$y(x) = a\exp(-bx) \tag{15.4.21}$$

can be rewritten as

$$\log[y(x)] = c - bx \tag{15.4.22}$$

which is linear in its parameters c and b. (Of course you must be aware that such transformations do not exactly take Gaussian errors into Gaussian errors.)

Also watch out for "non-parameters," as in

$$y(x) = a\exp(-bx + d) \tag{15.4.23}$$

Here the parameters a and d are, in fact, indistinguishable. This is a good example of where the normal equations will be exactly singular, and where SVD will find a zero singular value. SVD will then make a "least-squares" choice for setting a balance between a and d (or, rather, their equivalents in the linear model derived by taking the logarithms). However — and this is true whenever SVD returns a zero singular value — you are better advised to figure out analytically where the degeneracy is among your basis functions, and then make appropriate deletions in the basis set.

Here are two examples for user-supplied routines `funcs`. The first one is trivial and fits a general polynomial to a set of data:

```
SUBROUTINE fpoly(x,p,np)
INTEGER np
REAL x,p(np)
    Fitting routine for a polynomial of degree np-1, with np coefficients.
INTEGER j
p(1)=1.
do 11 j=2,np
    p(j)=p(j-1)*x
enddo 11
return
END
```

The second example is slightly less trivial. It is used to fit Legendre polynomials up to some order `nl-1` through a data set.

```
SUBROUTINE fleg(x,pl,nl)
INTEGER nl
REAL x,pl(nl)
    Fitting routine for an expansion with nl Legendre polynomials pl, evaluated using the
    recurrence relation as in §5.5.
INTEGER j
REAL d,f1,f2,twox
pl(1)=1.
pl(2)=x
if(nl.gt.2) then
    twox=2.*x
    f2=x
    d=1.
```

```
    do 11 j=3,nl
        f1=d
        f2=f2+twox
        d=d+1.
        pl(j)=(f2*pl(j-1)-f1*pl(j-2))/d
    enddo 11
endif
return
END
```

Multidimensional Fits

If you are measuring a single variable y as a function of more than one variable — say, a *vector* of variables $\mathbf{x}$, then your basis functions will be functions of a vector, $X_1(\mathbf{x}), \ldots, X_M(\mathbf{x})$. The χ^2 merit function is now

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(\mathbf{x}_i)}{\sigma_i} \right]^2 \tag{15.4.24}$$

All of the preceding discussion goes through unchanged, with x replaced by $\mathbf{x}$. In fact, if you are willing to tolerate a bit of programming hack, you can use the above programs without any modification: In both `lfit` and `svdfit`, the only use made of the array elements `x(i)` is that each element is in turn passed to the user-supplied routine `funcs`, which duly returns the values of the basis functions at that point. If you set `x(i)=i` before calling `lfit` or `svdfit`, and independently provide `funcs` with the true vector values of your data points (e.g., in a `COMMON` block), then `funcs` can translate from the fictitious `x(i)`'s to the actual data points before doing its work.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapters 8–9.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

15.5 Nonlinear Models

We now consider fitting when the model depends *nonlinearly* on the set of M unknown parameters $a_k, k = 1, 2, \ldots, M$. We use the same approach as in previous sections, namely to define a χ^2 merit function and determine best-fit parameters by its minimization. With nonlinear dependences, however, the minimization must proceed iteratively. Given trial values for the parameters, we develop a procedure that improves the trial solution. The procedure is then repeated until χ^2 stops (or effectively stops) decreasing.

```
    do 11 j=3,nl
        f1=d
        f2=f2+twox
        d=d+1.
        pl(j)=(f2*pl(j-1)-f1*pl(j-2))/d
    enddo 11
endif
return
END
```

Multidimensional Fits

If you are measuring a single variable y as a function of more than one variable — say, a *vector* of variables $\mathbf{x}$, then your basis functions will be functions of a vector, $X_1(\mathbf{x}), \ldots, X_M(\mathbf{x})$. The χ^2 merit function is now

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(\mathbf{x}_i)}{\sigma_i} \right]^2 \tag{15.4.24}$$

All of the preceding discussion goes through unchanged, with x replaced by $\mathbf{x}$. In fact, if you are willing to tolerate a bit of programming hack, you can use the above programs without any modification: In both `lfit` and `svdfit`, the only use made of the array elements `x(i)` is that each element is in turn passed to the user-supplied routine `funcs`, which duly returns the values of the basis functions at that point. If you set `x(i)=i` before calling `lfit` or `svdfit`, and independently provide `funcs` with the true vector values of your data points (e.g., in a `COMMON` block), then `funcs` can translate from the fictitious `x(i)`'s to the actual data points before doing its work.

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapters 8–9.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

15.5 Nonlinear Models

We now consider fitting when the model depends *nonlinearly* on the set of M unknown parameters $a_k, k = 1, 2, \ldots, M$. We use the same approach as in previous sections, namely to define a χ^2 merit function and determine best-fit parameters by its minimization. With nonlinear dependences, however, the minimization must proceed iteratively. Given trial values for the parameters, we develop a procedure that improves the trial solution. The procedure is then repeated until χ^2 stops (or effectively stops) decreasing.

How is this problem different from the general nonlinear function minimization problem already dealt with in Chapter 10? Superficially, not at all: Sufficiently close to the minimum, we expect the χ^2 function to be well approximated by a quadratic form, which we can write as

$$\chi^2(\mathbf{a}) \approx \gamma - \mathbf{d} \cdot \mathbf{a} + \frac{1}{2}\mathbf{a} \cdot \mathbf{D} \cdot \mathbf{a} \tag{15.5.1}$$

where $\mathbf{d}$ is an M-vector and $\mathbf{D}$ is an $M \times M$ matrix. (Compare equation 10.6.1.) If the approximation is a good one, we know how to jump from the current trial parameters $\mathbf{a}_{\rm cur}$ to the minimizing ones $\mathbf{a}_{\rm min}$ in a single leap, namely

$$\mathbf{a}_{\rm min} = \mathbf{a}_{\rm cur} + \mathbf{D}^{-1} \cdot \left[-\nabla\chi^2(\mathbf{a}_{\rm cur})\right] \tag{15.5.2}$$

(Compare equation 10.7.4.)

On the other hand, (15.5.1) might be a poor local approximation to the shape of the function that we are trying to minimize at $\mathbf{a}_{\rm cur}$. In that case, about all we can do is take a step down the gradient, as in the steepest descent method (§10.6). In other words,

$$\mathbf{a}_{\rm next} = \mathbf{a}_{\rm cur} - \text{constant} \times \nabla\chi^2(\mathbf{a}_{\rm cur}) \tag{15.5.3}$$

where the constant is small enough not to exhaust the downhill direction.

To use (15.5.2) or (15.5.3), we must be able to compute the gradient of the χ^2 function at any set of parameters $\mathbf{a}$. To use (15.5.2) we also need the matrix $\mathbf{D}$, which is the second derivative matrix (Hessian matrix) of the χ^2 merit function, at any $\mathbf{a}$.

Now, this is the crucial difference from Chapter 10: There, we had no way of directly evaluating the Hessian matrix. We were given only the ability to evaluate the function to be minimized and (in some cases) its gradient. Therefore, we had to resort to iterative methods *not just* because our function was nonlinear, *but also* in order to build up information about the Hessian matrix. Sections 10.7 and 10.6 concerned themselves with two different techniques for building up this information.

Here, life is much simpler. We *know* exactly the form of χ^2, since it is based on a model function that we ourselves have specified. Therefore the Hessian matrix is known to us. Thus we are free to use (15.5.2) whenever we care to do so. The only reason to use (15.5.3) will be failure of (15.5.2) to improve the fit, signaling failure of (15.5.1) as a good local approximation.

Calculation of the Gradient and Hessian

The model to be fitted is

$$y = y(x; \mathbf{a}) \tag{15.5.4}$$

and the χ^2 merit function is

$$\chi^2(\mathbf{a}) = \sum_{i=1}^{N} \left[\frac{y_i - y(x_i; \mathbf{a})}{\sigma_i}\right]^2 \tag{15.5.5}$$

The gradient of χ^2 with respect to the parameters $\mathbf{a}$, which will be zero at the χ^2 minimum, has components

$$\frac{\partial \chi^2}{\partial a_k} = -2 \sum_{i=1}^{N} \frac{[y_i - y(x_i;\mathbf{a})]}{\sigma_i^2} \frac{\partial y(x_i;\mathbf{a})}{\partial a_k} \qquad k = 1, 2, \ldots, M \tag{15.5.6}$$

Taking an additional partial derivative gives

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} = 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i;\mathbf{a})}{\partial a_k} \frac{\partial y(x_i;\mathbf{a})}{\partial a_l} - [y_i - y(x_i;\mathbf{a})] \frac{\partial^2 y(x_i;\mathbf{a})}{\partial a_l \partial a_k} \right] \tag{15.5.7}$$

It is conventional to remove the factors of 2 by defining

$$\beta_k \equiv -\frac{1}{2} \frac{\partial \chi^2}{\partial a_k} \qquad \alpha_{kl} \equiv \frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_k \partial a_l} \tag{15.5.8}$$

making $[\alpha] = \frac{1}{2}\mathbf{D}$ in equation (15.5.2), in terms of which that equation can be rewritten as the set of linear equations

$$\sum_{l=1}^{M} \alpha_{kl} \, \delta a_l = \beta_k \tag{15.5.9}$$

This set is solved for the increments δa_l that, added to the current approximation, give the next approximation. In the context of least-squares, the matrix $[\alpha]$, equal to one-half times the Hessian matrix, is usually called the *curvature matrix*.

Equation (15.5.3), the steepest descent formula, translates to

$$\delta a_l = \text{constant} \times \beta_l \tag{15.5.10}$$

Note that the components α_{kl} of the Hessian matrix (15.5.7) depend both on the first derivatives and on the second derivatives of the basis functions with respect to their parameters. Some treatments proceed to ignore the second derivative without comment. We will ignore it also, but only *after* a few comments.

Second derivatives occur because the gradient (15.5.6) already has a dependence on $\partial y / \partial a_k$, so the next derivative simply must contain terms involving $\partial^2 y / \partial a_l \partial a_k$. The second derivative term can be dismissed when it is zero (as in the linear case of equation 15.4.8), or small enough to be negligible when compared to the term involving the first derivative. It also has an additional possibility of being ignorably small in practice: The term multiplying the second derivative in equation (15.5.7) is $[y_i - y(x_i;\mathbf{a})]$. For a successful model, this term should just be the random measurement error of each point. This error can have either sign, and should in general be uncorrelated with the model. Therefore, the second derivative terms tend to cancel out when summed over i.

Inclusion of the second-derivative term can in fact be destabilizing if the model fits badly or is contaminated by outlier points that are unlikely to be offset by

compensating points of opposite sign. From this point on, we will always use as the definition of α_{kl} the formula

$$\alpha_{kl} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[\frac{\partial y(x_i;\mathbf{a})}{\partial a_k} \frac{\partial y(x_i;\mathbf{a})}{\partial a_l} \right] \tag{15.5.11}$$

This expression more closely resembles its linear cousin (15.4.8). You should understand that minor (or even major) fiddling with $[\alpha]$ has no effect at all on what final set of parameters **a** is reached, but affects only the iterative route that is taken in getting there. The condition at the χ^2 minimum, that $\beta_k = 0$ for all k, is independent of how $[\alpha]$ is defined.

Levenberg-Marquardt Method

Marquardt [1] has put forth an elegant method, related to an earlier suggestion of Levenberg, for varying smoothly between the extremes of the inverse-Hessian method (15.5.9) and the steepest descent method (15.5.10). The latter method is used far from the minimum, switching continuously to the former as the minimum is approached. This *Levenberg-Marquardt method* (also called *Marquardt method*) works very well in practice and has become the standard of nonlinear least-squares routines.

The method is based on two elementary, but important, insights. Consider the "constant" in equation (15.5.10). What should it be, even in order of magnitude? What sets its scale? There is no information about the answer in the gradient. That tells only the slope, not how far that slope extends. Marquardt's first insight is that the components of the Hessian matrix, even if they are not usable in any precise fashion, give *some* information about the order-of-magnitude scale of the problem.

The quantity χ^2 is nondimensional, i.e., is a pure number; this is evident from its definition (15.5.5). On the other hand, β_k has the dimensions of $1/a_k$, which may well be dimensional, i.e., have units like cm^{-1}, or kilowatt-hours, or whatever. (In fact, each component of β_k can have different dimensions!) The constant of proportionality between β_k and δa_k must therefore have the dimensions of a_k^2. Scan the components of $[\alpha]$ and you see that there is only one obvious quantity with these dimensions, and that is $1/\alpha_{kk}$, the reciprocal of the diagonal element. So that must set the scale of the constant. But that scale might itself be too big. So let's divide the constant by some (nondimensional) fudge factor λ, with the possibility of setting $\lambda \gg 1$ to cut down the step. In other words, replace equation (15.5.10) by

$$\delta a_l = \frac{1}{\lambda \alpha_{ll}} \beta_l \qquad \text{or} \qquad \lambda\, \alpha_{ll}\, \delta a_l = \beta_l \tag{15.5.12}$$

It is necessary that a_{ll} be positive, but this is guaranteed by definition (15.5.11) — another reason for adopting that equation.

Marquardt's second insight is that equations (15.5.12) and (15.5.9) can be combined if we define a new matrix α' by the following prescription

$$\begin{aligned} \alpha'_{jj} &\equiv \alpha_{jj}(1+\lambda) \\ \alpha'_{jk} &\equiv \alpha_{jk} \qquad (j \neq k) \end{aligned} \tag{15.5.13}$$

and then replace both (15.5.12) and (15.5.9) by

$$\sum_{l=1}^{M} \alpha'_{kl}\,\delta a_l = \beta_k \tag{15.5.14}$$

When λ is very large, the matrix α' is forced into being *diagonally dominant*, so equation (15.5.14) goes over to be identical to (15.5.12). On the other hand, as λ approaches zero, equation (15.5.14) goes over to (15.5.9).

Given an initial guess for the set of fitted parameters **a**, the recommended Marquardt recipe is as follows:

- Compute $\chi^2(\mathbf{a})$.
- Pick a modest value for λ, say $\lambda = 0.001$.
- (†) Solve the linear equations (15.5.14) for $\delta\mathbf{a}$ and evaluate $\chi^2(\mathbf{a}+\delta\mathbf{a})$.
- If $\chi^2(\mathbf{a}+\delta\mathbf{a}) \geq \chi^2(\mathbf{a})$, *increase* λ by a factor of 10 (or any other substantial factor) and go back to (†).
- If $\chi^2(\mathbf{a}+\delta\mathbf{a}) < \chi^2(\mathbf{a})$, *decrease* λ by a factor of 10, update the trial solution $\mathbf{a} \leftarrow \mathbf{a}+\delta\mathbf{a}$, and go back to (†).

Also necessary is a condition for stopping. Iterating to convergence (to machine accuracy or to the roundoff limit) is generally wasteful and unnecessary since the minimum is at best only a statistical estimate of the parameters **a**. As we will see in §15.6, a change in the parameters that changes χ^2 by an amount $\ll 1$ is *never* statistically meaningful.

Furthermore, it is not uncommon to find the parameters wandering around near the minimum in a flat valley of complicated topography. The reason is that Marquardt's method generalizes the method of normal equations (§15.4), hence has the same problem as that method with regard to near-degeneracy of the minimum. Outright failure by a zero pivot is possible, but unlikely. More often, a small pivot will generate a large correction which is then rejected, the value of λ being then increased. For sufficiently large λ the matrix $[\alpha']$ is positive definite and can have no small pivots. Thus the method does tend to stay away from zero pivots, but at the cost of a tendency to wander around doing steepest descent in very un-steep degenerate valleys.

These considerations suggest that, in practice, one might as well stop iterating on the first or second occasion that χ^2 decreases by a negligible amount, say either less than 0.01 absolutely or (in case roundoff prevents that being reached) some fractional amount like 10^{-3}. Don't stop after a step where χ^2 *increases*: That only shows that λ has not yet adjusted itself optimally.

Once the acceptable minimum has been found, one wants to set $\lambda = 0$ and compute the matrix

$$[C] \equiv [\alpha]^{-1} \tag{15.5.15}$$

which, as before, is the estimated covariance matrix of the standard errors in the fitted parameters **a** (see next section).

The following pair of subroutines encodes Marquardt's method for nonlinear parameter estimation. Much of the organization matches that used in `lfit` of §15.4. In particular the array `ia(1:ma)` must be input with components one or zero corresponding to whether the respective parameter values `a(1:ma)` are to be fitted for or held fixed at their input values, respectively.

The routine mrqmin performs one iteration of Marquardt's method. It is first called (once) with alamda < 0, which signals the routine to initialize. alamda is returned on the first and all subsequent calls as the suggested value of λ for the next iteration; a and chisq are always returned as the best parameters found so far and their χ^2. When convergence is deemed satisfactory, set alamda to zero before a final call. The matrices alpha and covar (which were used as workspace in all previous calls) will then be set to the curvature and covariance matrices for the converged parameter values. The arguments alpha, a, and chisq must not be modified between calls, nor should alamda be, except to set it to zero for the final call. When an uphill step is taken, chisq and a are returned with their input (best) values, but alamda is returned with an increased value.

The routine mrqmin calls the routine mrqcof for the computation of the matrix $[\alpha]$ (equation 15.5.11) and vector β (equations 15.5.6 and 15.5.8). In turn mrqcof calls the user-supplied routine funcs(x,a,y,dyda), which for input values x $\equiv x_i$ and a $\equiv \mathbf{a}$ returns the model function y $\equiv y(x_i;\mathbf{a})$ and the vector of derivatives dyda $\equiv \partial y/\partial a_k$.

```
      SUBROUTINE mrqmin(x,y,sig,ndata,a,ia,ma,covar,alpha,nca,
     *     chisq,funcs,alamda)
      INTEGER ma,nca,ndata,ia(ma),MMAX
      REAL alamda,chisq,funcs,a(ma),alpha(nca,nca),covar(nca,nca),
     *     sig(ndata),x(ndata),y(ndata)
      PARAMETER (MMAX=20)                          Set to largest number of fit parameters.
C     USES covsrt,gaussj,mrqcof
        Levenberg-Marquardt method, attempting to reduce the value χ² of a fit between a set of
        data points x(1:ndata), y(1:ndata) with individual standard deviations sig(1:ndata),
        and a nonlinear function dependent on ma coefficients a(1:ma). The input array ia(1:ma)
        indicates by nonzero entries those components of a that should be fitted for, and by zero
        entries those components that should be held fixed at their input values. The program
        returns current best-fit values for the parameters a(1:ma), and χ² = chisq. The ar-
        rays covar(1:nca,1:nca), alpha(1:nca,1:nca) with physical dimension nca (≥ the
        number of fitted parameters) are used as working space during most iterations. Supply a
        subroutine funcs(x,a,yfit,dyda,ma) that evaluates the fitting function yfit, and its
        derivatives dyda with respect to the fitting parameters a at x. On the first call provide
        an initial guess for the parameters a, and set alamda<0 for initialization (which then sets
        alamda=.001). If a step succeeds chisq becomes smaller and alamda decreases by a
        factor of 10. If a step fails alamda grows by a factor of 10. You must call this routine
        repeatedly until convergence is achieved. Then, make one final call with alamda=0, so
        that covar(1:ma,1:ma) returns the covariance matrix, and alpha the curvature matrix.
        (Parameters held fixed will return zero covariances.)
      INTEGER j,k,l,mfit
      REAL ochisq,atry(MMAX),beta(MMAX),da(MMAX)
      SAVE ochisq,atry,beta,da,mfit
      if(alamda.lt.0.)then                         Initialization.
        mfit=0
        do 11 j=1,ma
          if (ia(j).ne.0) mfit=mfit+1
        enddo 11
        alamda=0.001
        call mrqcof(x,y,sig,ndata,a,ia,ma,alpha,beta,nca,chisq,funcs)
        ochisq=chisq
        do 12 j=1,ma
          atry(j)=a(j)
        enddo 12
      endif
      do 14 j=1,mfit                               Alter linearized fitting matrix, by augmenting
        do 13 k=1,mfit                               diagonal elements.
```

```
            covar(j,k)=alpha(j,k)
        enddo 13
        covar(j,j)=alpha(j,j)*(1.+alamda)
        da(j)=beta(j)
    enddo 14
    call gaussj(covar,mfit,nca,da,1,1)          Matrix solution.
    if(alamda.eq.0.)then                        Once converged, evaluate covariance matrix.
        call covsrt(covar,nca,ma,ia,mfit)
        call covsrt(alpha,nca,ma,ia,mfit)       Spread out alpha to its full size too.
        return
    endif
    j=0
    do 15 l=1,ma                                Did the trial succeed?
        if(ia(l).ne.0) then
            j=j+1
            atry(l)=a(l)+da(j)
        endif
    enddo 15
    call mrqcof(x,y,sig,ndata,atry,ia,ma,covar,da,nca,chisq,funcs)
    if(chisq.lt.ochisq)then                     Success, accept the new solution.
        alamda=0.1*alamda
        ochisq=chisq
        do 17 j=1,mfit
            do 16 k=1,mfit
                alpha(j,k)=covar(j,k)
            enddo 16
            beta(j)=da(j)
        enddo 17
        do 18 l=1,ma
            a(l)=atry(l)
        enddo 18
    else                                        Failure, increase alamda and return.
        alamda=10.*alamda
        chisq=ochisq
    endif
    return
    END
```

Notice the use of the routine `covsrt` from §15.4. This is merely for rearranging the covariance matrix `covar` into the order of all `ma` parameters. The above routine also makes use of

```
    SUBROUTINE mrqcof(x,y,sig,ndata,a,ia,ma,alpha,beta,nalp,
*       chisq,funcs)
    INTEGER ma,nalp,ndata,ia(ma),MMAX
    REAL chisq,a(ma),alpha(nalp,nalp),beta(ma),sig(ndata),x(ndata),
*       y(ndata)
    EXTERNAL funcs
    PARAMETER (MMAX=20)
      Used by mrqmin to evaluate the linearized fitting matrix alpha, and vector beta as in
      (15.5.8), and calculate χ².
    INTEGER mfit,i,j,k,l,m
    REAL dy,sig2i,wt,ymod,dyda(MMAX)
    mfit=0
    do 11 j=1,ma
        if (ia(j).ne.0) mfit=mfit+1
    enddo 11
    do 13 j=1,mfit                              Initialize (symmetric) alpha, beta.
        do 12 k=1,j
```

```
        alpha(j,k)=0.
    enddo 12
    beta(j)=0.
enddo 13
chisq=0.
do 16 i=1,ndata                    Summation loop over all data.
    call funcs(x(i),a,ymod,dyda,ma)
    sig2i=1./(sig(i)*sig(i))
    dy=y(i)-ymod
    j=0
    do 15 l=1,ma
        if(ia(l).ne.0) then
            j=j+1
            wt=dyda(l)*sig2i
            k=0
            do 14 m=1,l
                if(ia(m).ne.0) then
                    k=k+1
                    alpha(j,k)=alpha(j,k)+wt*dyda(m)
                endif
            enddo 14
            beta(j)=beta(j)+dy*wt
        endif
    enddo 15
    chisq=chisq+dy*dy*sig2i        And find χ².
enddo 16
do 18 j=2,mfit                     Fill in the symmetric side.
    do 17 k=1,j-1
        alpha(k,j)=alpha(j,k)
    enddo 17
enddo 18
return
END
```

Example

The following subroutine `fgauss` is an example of a user-supplied subroutine `funcs`. Used with the above routine `mrqmin` (in turn using `mrqcof`, `covsrt`, and `gaussj`), it fits for the model

$$y(x) = \sum_{k=1}^{K} B_k \exp\left[-\left(\frac{x - E_k}{G_k}\right)^2\right] \tag{15.5.16}$$

which is a sum of K Gaussians, each having a variable position, amplitude, and width. We store the parameters in the order $B_1, E_1, G_1, B_2, E_2, G_2, \ldots, B_K, E_K, G_K$.

```
SUBROUTINE fgauss(x,a,y,dyda,na)
INTEGER na
REAL x,y,a(na),dyda(na)
```

$y(x;a)$ is the sum of na/3 Gaussians (15.5.16). The amplitude, center, and width of the Gaussians are stored in consecutive locations of a: a(i) $= B_k$, a(i+1) $= E_k$, a(i+2) $= G_k$, $k = 1, ..., $ na/3.

```
INTEGER i
REAL arg,ex,fac
y=0.
do 11 i=1,na-1,3
    arg=(x-a(i+1))/a(i+2)
    ex=exp(-arg**2)
    fac=a(i)*ex*2.*arg
    y=y+a(i)*ex
    dyda(i)=ex
    dyda(i+1)=fac/a(i+2)
    dyda(i+2)=fac*arg/a(i+2)
enddo 11
return
END
```

More Advanced Methods for Nonlinear Least Squares

The Levenberg-Marquardt algorithm can be implemented as a model-trust region method for minimization (see §9.7 and ref. [2]) applied to the special case of a least squares function. A code of this kind due to Moré [3] can be found in MINPACK [4]. Another algorithm for nonlinear least-squares keeps the second-derivative term we dropped in the Levenberg-Marquardt method whenever it would be better to do so. These methods are called "full Newton-type" methods and are reputed to be more robust than Levenberg-Marquardt, but more complex. One implementation is the code NL2SOL [5].

CITED REFERENCES AND FURTHER READING:

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill), Chapter 11.

Marquardt, D.W. 1963, *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, pp. 431–441. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.2 (by J.E. Dennis).

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Moré, J.J. 1977, in *Numerical Analysis*, Lecture Notes in Mathematics, vol. 630, G.A. Watson, ed. (Berlin: Springer-Verlag), pp. 105–116. [3]

Moré, J.J., Garbow, B.S., and Hillstrom, K.E. 1980, *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74. [4]

Dennis, J.E., Gay, D.M, and Welsch, R.E. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 348–368; *op. cit.*, pp. 369–383. [5].

15.6 Confidence Limits on Estimated Model Parameters

Several times already in this chapter we have made statements about the standard errors, or uncertainties, in a set of M estimated parameters $\mathbf{a}$. We have given some formulas for computing standard deviations or variances of individual parameters (equations 15.2.9, 15.4.15, 15.4.19), as well as some formulas for covariances between pairs of parameters (equation 15.2.10; remark following equation 15.4.15; equation 15.4.20; equation 15.5.15).

In this section, we want to be more explicit regarding the precise meaning of these quantitative uncertainties, and to give further information about how quantitative confidence limits on fitted parameters can be estimated. The subject can get somewhat technical, and even somewhat confusing, so we will try to make precise statements, even when they must be offered without proof.

Figure 15.6.1 shows the conceptual scheme of an experiment that "measures" a set of parameters. There is some underlying true set of parameters $\mathbf{a}_{\rm true}$ that are known to Mother Nature but hidden from the experimenter. These true parameters are statistically realized, along with random measurement errors, as a measured data set, which we will symbolize as $\mathcal{D}_{(0)}$. The data set $\mathcal{D}_{(0)}$ *is* known to the experimenter. He or she fits the data to a model by χ^2 minimization or some other technique, and obtains measured, i.e., fitted, values for the parameters, which we here denote $\mathbf{a}_{(0)}$.

Because measurement errors have a random component, $\mathcal{D}_{(0)}$ is not a unique realization of the true parameters $\mathbf{a}_{\rm true}$. Rather, there are infinitely many other realizations of the true parameters as "hypothetical data sets" each of which *could* have been the one measured, but happened not to be. Let us symbolize these by $\mathcal{D}_{(1)}, \mathcal{D}_{(2)}, \ldots$. Each one, had it been realized, would have given a slightly different set of fitted parameters, $\mathbf{a}_{(1)}, \mathbf{a}_{(2)}, \ldots$, respectively. These parameter sets $\mathbf{a}_{(i)}$ therefore occur with some probability distribution in the M-dimensional space of all possible parameter sets $\mathbf{a}$. The actual measured set $\mathbf{a}_{(0)}$ is one member drawn from this distribution.

Even more interesting than the probability distribution of $\mathbf{a}_{(i)}$ would be the distribution of the difference $\mathbf{a}_{(i)} - \mathbf{a}_{\rm true}$. This distribution differs from the former one by a translation that puts Mother Nature's true value at the origin. If we knew *this* distribution, we would know everything that there is to know about the quantitative uncertainties in our experimental measurement $\mathbf{a}_{(0)}$.

So the name of the game is to find some way of estimating or approximating the probability distribution of $\mathbf{a}_{(i)} - \mathbf{a}_{\rm true}$ without knowing $\mathbf{a}_{\rm true}$ and without having available to us an infinite universe of hypothetical data sets.

Monte Carlo Simulation of Synthetic Data Sets

Although the measured parameter set $\mathbf{a}_{(0)}$ is not the true one, let us consider a fictitious world in which it *was* the true one. Since we hope that our measured parameters are not *too* wrong, we hope that that fictitious world is not too different from the actual world with parameters $\mathbf{a}_{\rm true}$. In particular, let us hope — no, let us *assume* — that the shape of the probability distribution $\mathbf{a}_{(i)} - \mathbf{a}_{(0)}$ in the fictitious world is the same, or very nearly the same, as the shape of the probability distribution

15.6 Confidence Limits on Estimated Model Parameters

Several times already in this chapter we have made statements about the standard errors, or uncertainties, in a set of M estimated parameters $\mathbf{a}$. We have given some formulas for computing standard deviations or variances of individual parameters (equations 15.2.9, 15.4.15, 15.4.19), as well as some formulas for covariances between pairs of parameters (equation 15.2.10; remark following equation 15.4.15; equation 15.4.20; equation 15.5.15).

In this section, we want to be more explicit regarding the precise meaning of these quantitative uncertainties, and to give further information about how quantitative confidence limits on fitted parameters can be estimated. The subject can get somewhat technical, and even somewhat confusing, so we will try to make precise statements, even when they must be offered without proof.

Figure 15.6.1 shows the conceptual scheme of an experiment that "measures" a set of parameters. There is some underlying true set of parameters $\mathbf{a}_{\rm true}$ that are known to Mother Nature but hidden from the experimenter. These true parameters are statistically realized, along with random measurement errors, as a measured data set, which we will symbolize as $\mathcal{D}_{(0)}$. The data set $\mathcal{D}_{(0)}$ *is* known to the experimenter. He or she fits the data to a model by χ^2 minimization or some other technique, and obtains measured, i.e., fitted, values for the parameters, which we here denote $\mathbf{a}_{(0)}$.

Because measurement errors have a random component, $\mathcal{D}_{(0)}$ is not a unique realization of the true parameters $\mathbf{a}_{\rm true}$. Rather, there are infinitely many other realizations of the true parameters as "hypothetical data sets" each of which *could* have been the one measured, but happened not to be. Let us symbolize these by $\mathcal{D}_{(1)}, \mathcal{D}_{(2)}, \ldots$. Each one, had it been realized, would have given a slightly different set of fitted parameters, $\mathbf{a}_{(1)}, \mathbf{a}_{(2)}, \ldots$, respectively. These parameter sets $\mathbf{a}_{(i)}$ therefore occur with some probability distribution in the M-dimensional space of all possible parameter sets $\mathbf{a}$. The actual measured set $\mathbf{a}_{(0)}$ is one member drawn from this distribution.

Even more interesting than the probability distribution of $\mathbf{a}_{(i)}$ would be the distribution of the difference $\mathbf{a}_{(i)} - \mathbf{a}_{\rm true}$. This distribution differs from the former one by a translation that puts Mother Nature's true value at the origin. If we knew *this* distribution, we would know everything that there is to know about the quantitative uncertainties in our experimental measurement $\mathbf{a}_{(0)}$.

So the name of the game is to find some way of estimating or approximating the probability distribution of $\mathbf{a}_{(i)} - \mathbf{a}_{\rm true}$ without knowing $\mathbf{a}_{\rm true}$ and without having available to us an infinite universe of hypothetical data sets.

Monte Carlo Simulation of Synthetic Data Sets

Although the measured parameter set $\mathbf{a}_{(0)}$ is not the true one, let us consider a fictitious world in which it *was* the true one. Since we hope that our measured parameters are not *too* wrong, we hope that that fictitious world is not too different from the actual world with parameters $\mathbf{a}_{\rm true}$. In particular, let us hope — no, let us *assume* — that the shape of the probability distribution $\mathbf{a}_{(i)} - \mathbf{a}_{(0)}$ in the fictitious world is the same, or very nearly the same, as the shape of the probability distribution

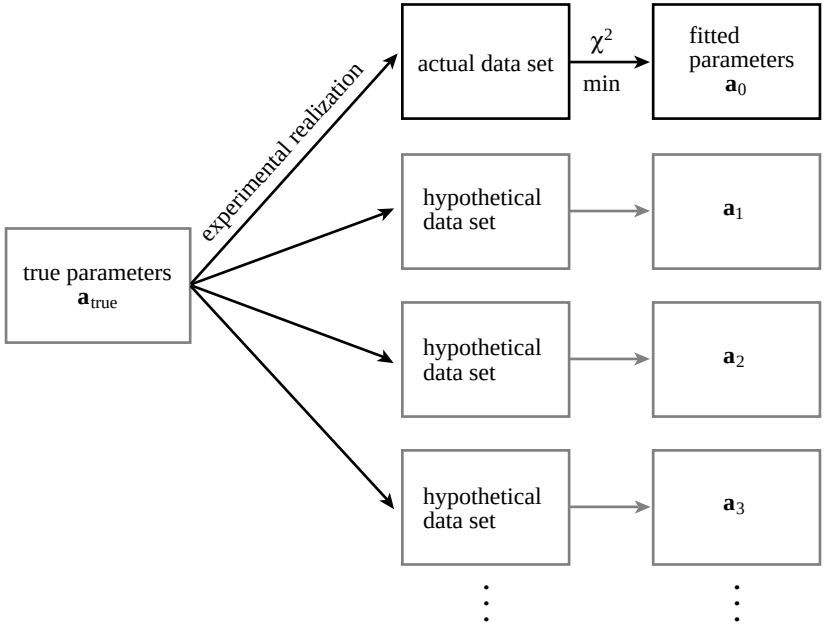


Figure 15.6.1. A statistical universe of data sets from an underlying model. True parameters $\mathbf{a}_{\rm true}$ are realized in a data set, from which fitted (observed) parameters $\mathbf{a}_0$ are obtained. If the experiment were repeated many times, new data sets and new values of the fitted parameters would be obtained.

$\mathbf{a}_{(i)} - \mathbf{a}_{\rm true}$ in the real world. Notice that we are not assuming that $\mathbf{a}_{(0)}$ and $\mathbf{a}_{\rm true}$ are equal; they are certainly not. We are only assuming that the way in which random errors enter the experiment and data analysis does not vary rapidly as a function of $\mathbf{a}_{\rm true}$, so that $\mathbf{a}_{(0)}$ can serve as a reasonable surrogate.

Now, often, the distribution of $\mathbf{a}_{(i)} - \mathbf{a}_{(0)}$ in the fictitious world *is* within our power to calculate (see Figure 15.6.2). If we know something about the process that generated our data, given an assumed set of parameters $\mathbf{a}_{(0)}$, then we can usually figure out how to *simulate* our own sets of "synthetic" realizations of these parameters as "synthetic data sets." The procedure is to draw random numbers from appropriate distributions (cf. §7.2–§7.3) so as to mimic our best understanding of the underlying process and measurement errors in our apparatus. With such random draws, we construct data sets with exactly the same numbers of measured points, and precisely the same values of all control (independent) variables, as our actual data set $\mathcal{D}_{(0)}$. Let us call these simulated data sets $\mathcal{D}^S_{(1)}, \mathcal{D}^S_{(2)}, \ldots$. By construction these are supposed to have exactly the same statistical relationship to $\mathbf{a}_{(0)}$ as the $\mathcal{D}_{(i)}$'s have to $\mathbf{a}_{\rm true}$. (For the case where you don't know enough about what you are measuring to do a credible job of simulating it, see below.)

Next, for each $\mathcal{D}^S_{(j)}$, perform exactly the same procedure for estimation of parameters, e.g., χ^2 minimization, as was performed on the actual data to get the parameters $\mathbf{a}_{(0)}$, giving simulated measured parameters $\mathbf{a}^S_{(1)}, \mathbf{a}^S_{(2)}, \ldots$. Each simulated measured parameter set yields a point $\mathbf{a}^S_{(i)} - \mathbf{a}_{(0)}$. Simulate enough data sets and enough derived simulated measured parameters, and you map out the desired probability distribution in M dimensions.

In fact, the ability to do *Monte Carlo simulations* in this fashion has revo-

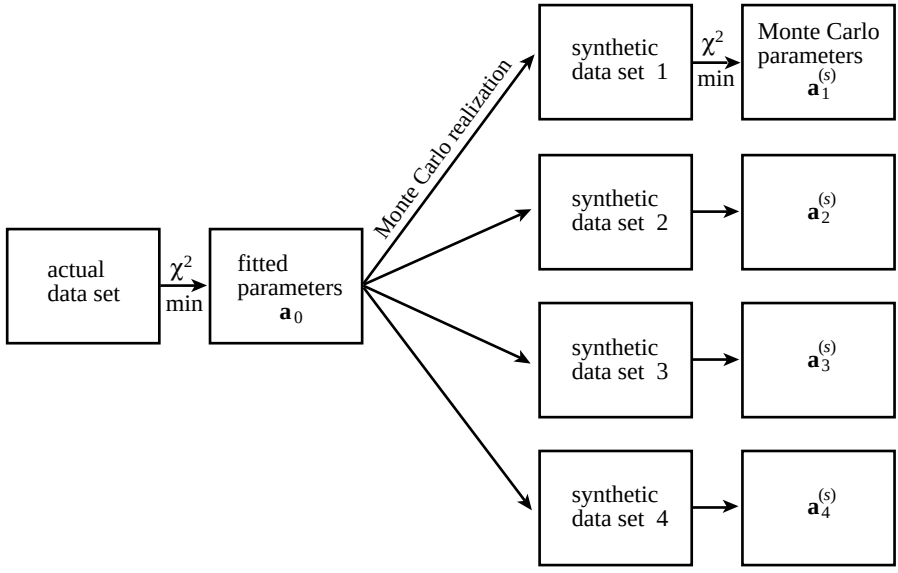


Figure 15.6.2. Monte Carlo simulation of an experiment. The fitted parameters from an actual experiment are used as surrogates for the true parameters. Computer-generated random numbers are used to simulate many synthetic data sets. Each of these is analyzed to obtain its fitted parameters. The distribution of these fitted parameters around the (known) surrogate true parameters is thus studied.

lutionized many fields of modern experimental science. Not only is one able to characterize the errors of parameter estimation in a very precise way; one can also try out on the computer different methods of parameter estimation, or different data reduction techniques, and seek to minimize the uncertainty of the result according to any desired criteria. Offered the choice between mastery of a five-foot shelf of analytical statistics books and middling ability at performing statistical Monte Carlo simulations, we would surely choose to have the latter skill.

Quick-and-Dirty Monte Carlo: The Bootstrap Method

Here is a powerful technique that can often be used when you don't know enough about the underlying process, or the nature of your measurement errors, to do a credible Monte Carlo simulation. Suppose that your data set consists of N *independent and identically distributed* (or *iid*) "data points." Each data point probably consists of several numbers, e.g., one or more control variables (uniformly distributed, say, in the range that you have decided to measure) and one or more associated measured values (each distributed however Mother Nature chooses). "Iid" means that the sequential order of the data points is not of consequence to the process that you are using to get the fitted parameters $\mathbf{a}$. For example, a χ^2 sum like (15.5.5) does not care in what order the points are added. Even simpler examples are the mean value of a measured quantity, or the mean of some function of the measured quantities.

The *bootstrap method* [1] uses the actual data set $\mathcal{D}^S_{(0)}$, with its N data points, to generate any number of synthetic data sets $\mathcal{D}^S_{(1)}, \mathcal{D}^S_{(2)}, \ldots$, also with N data points. The procedure is simply to draw N data points at a time *with replacement* from the

set $\mathcal{D}^S_{(0)}$. Because of the replacement, you do not simply get back your original data set each time. You get sets in which a random fraction of the original points, typically $\sim 1/e \approx 37\%$, are replaced by *duplicated* original points. Now, exactly as in the previous discussion, you subject these data sets to the same estimation procedure as was performed on the actual data, giving a set of simulated measured parameters $\mathbf{a}^S_{(1)}, \mathbf{a}^S_{(2)}, \ldots$. These will be distributed around $\mathbf{a}_{(0)}$ in close to the same way that $\mathbf{a}_{(0)}$ is distributed around $\mathbf{a}_{\rm true}$.

Sounds like getting something for nothing, doesn't it? In fact, it has taken more than a decade for the bootstrap method to become accepted by statisticians. By now, however, enough theorems have been proved to render the bootstrap reputable (see [2] for references). The basic idea behind the bootstrap is that the actual data set, viewed as a probability distribution consisting of delta functions at the measured values, is in most cases the best — or only — available estimator of the underlying probability distribution. It takes courage, but one can often simply use *that* distribution as the basis for Monte Carlo simulations.

Watch out for cases where the bootstrap's "iid" assumption is violated. For example, if you have made measurements at evenly spaced intervals of some control variable, then you can *usually* get away with pretending that these are "iid," uniformly distributed over the measured range. However, some estimators of **a** (e.g., ones involving Fourier methods) might be particularly sensitive to all the points on a grid being present. In that case, the bootstrap is going to give a wrong distribution. Also watch out for estimators that look at anything like small-scale clumpiness within the N data points, or estimators that sort the data and look at sequential differences. Obviously the bootstrap will fail on these, too. (The theorems justifying the method are still true, but some of their technical assumptions are violated by these examples.)

For a large class of problems, however, the bootstrap does yield easy, *very quick*, Monte Carlo estimates of the errors in an estimated parameter set.

Confidence Limits

Rather than present all details of the probability distribution of errors in parameter estimation, it is common practice to summarize the distribution in the form of *confidence limits*. The full probability distribution is a function defined on the M-dimensional space of parameters **a**. A *confidence region* (or *confidence interval*) is just a region of that M-dimensional space (hopefully a small region) that contains a certain (hopefully large) percentage of the total probability distribution. You point to a confidence region and say, e.g., "there is a 99 percent chance that the true parameter values fall within this region around the measured value."

It is worth emphasizing that you, the experimenter, get to pick both the *confidence level* (99 percent in the above example), and the shape of the confidence region. The only requirement is that your region does include the stated percentage of probability. Certain percentages are, however, customary in scientific usage: 68.3 percent (the lowest confidence worthy of quoting), 90 percent, 95.4 percent, 99 percent, and 99.73 percent. Higher confidence levels are conventionally "ninety-nine point nine ... nine." As for shape, obviously you want a region that is compact and reasonably centered on your measurement $\mathbf{a}_{(0)}$, since the whole purpose of a confidence limit is to inspire confidence in that measured value. In one dimension,

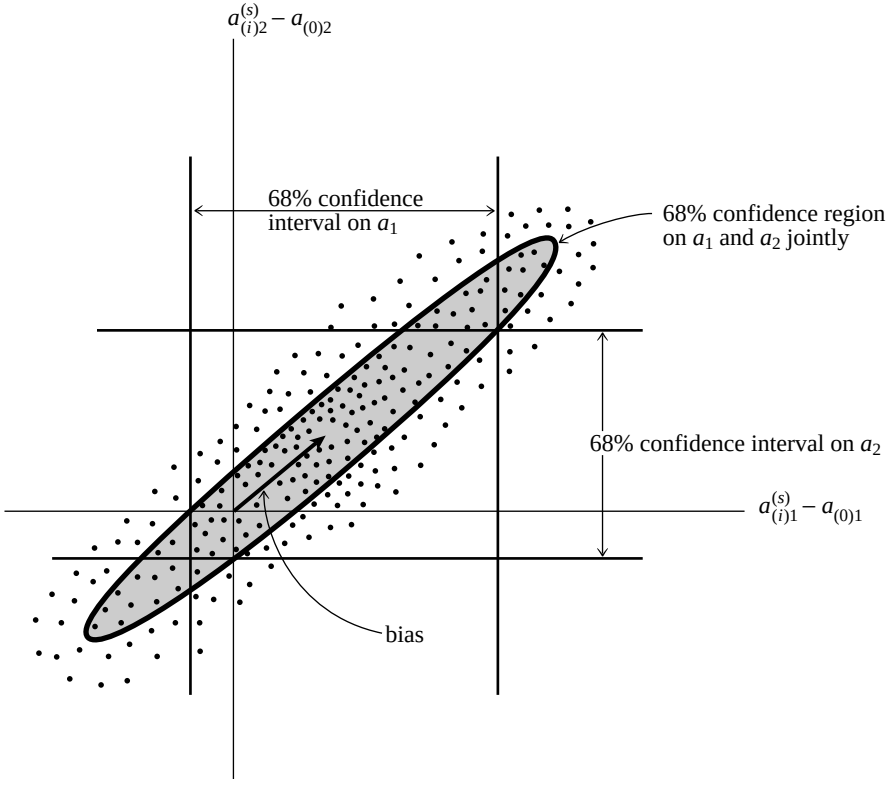


Figure 15.6.3. Confidence intervals in 1 and 2 dimensions. The same fraction of measured points (here 68%) lies (i) between the two vertical lines, (ii) between the two horizontal lines, (iii) within the ellipse.

the convention is to use a line segment centered on the measured value; in higher dimensions, ellipses or ellipsoids are most frequently used.

You might suspect, correctly, that the numbers 68.3 percent, 95.4 percent, and 99.73 percent, and the use of ellipsoids, have some connection with a normal distribution. That is true historically, but not always relevant nowadays. In general, the probability distribution of the parameters will not be normal, and the above numbers, used as levels of confidence, are purely matters of convention.

Figure 15.6.3 sketches a possible probability distribution for the case $M = 2$. Shown are three different confidence regions which might usefully be given, all at the same confidence level. The two vertical lines enclose a band (horizontal inverval) which represents the 68 percent confidence interval for the variable a_1 without regard to the value of a_2. Similarly the horizontal lines enclose a 68 percent confidence interval for a_2. The ellipse shows a 68 percent confidence interval for a_1 and a_2 jointly. Notice that to enclose the same probability as the two bands, the ellipse must necessarily extend outside of both of them (a point we will return to below).

Constant Chi-Square Boundaries as Confidence Limits

When the method used to estimate the parameters $\mathbf{a}_{(0)}$ is chi-square minimization, as in the previous sections of this chapter, then there is a natural choice for the

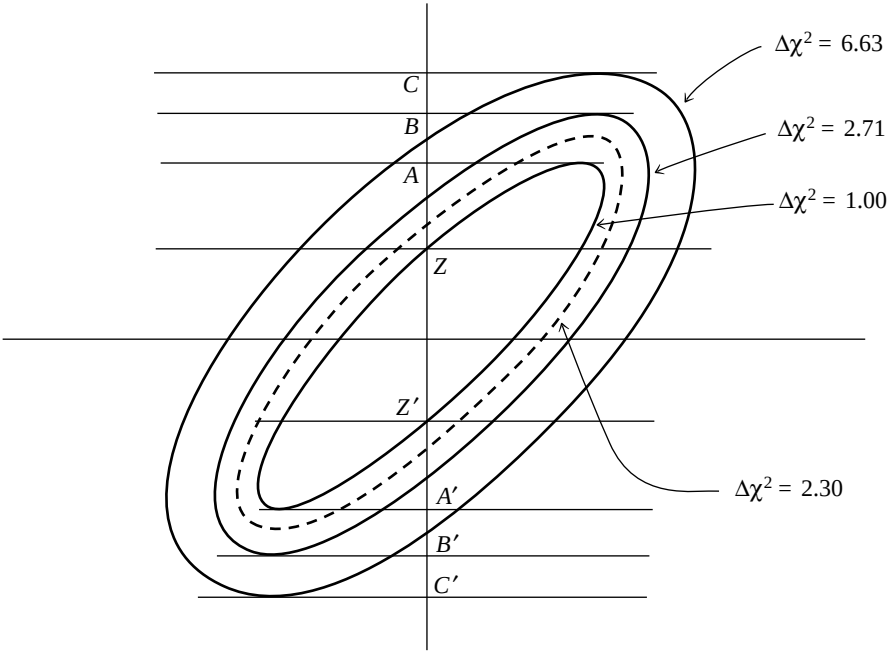


Figure 15.6.4. Confidence region ellipses corresponding to values of chi-square larger than the fitted minimum. The solid curves, with $\Delta\chi^2 = 1.00, 2.71, 6.63$ project onto one-dimensional intervals AA', BB', CC'. These intervals — not the ellipses themselves — contain 68.3%, 90%, and 99% of normally distributed data. The ellipse that contains 68.3% of normally distributed data is shown dashed, and has $\Delta\chi^2 = 2.30$. For additional numerical values, see accompanying table.

shape of confidence intervals, whose use is almost universal. For the observed data set $\mathcal{D}_{(0)}$, the value of χ^2 is a minimum at $\mathbf{a}_{(0)}$. Call this minimum value $\chi^2_{\min}$. If the vector $\mathbf{a}$ of parameter values is perturbed away from $\mathbf{a}_{(0)}$, then χ^2 increases. The region within which χ^2 increases by no more than a set amount $\Delta\chi^2$ defines some M-dimensional confidence region around $\mathbf{a}_{(0)}$. If $\Delta\chi^2$ is set to be a large number, this will be a big region; if it is small, it will be small. Somewhere in between there will be choices of $\Delta\chi^2$ that cause the region to contain, variously, 68 percent, 90 percent, etc. of probability distribution for $\mathbf{a}$'s, as defined above. These regions are taken as the confidence regions for the parameters $\mathbf{a}_{(0)}$.

Very frequently one is interested not in the full M-dimensional confidence region, but in individual confidence regions for some smaller number ν of parameters. For example, one might be interested in the confidence interval of each parameter taken separately (the bands in Figure 15.6.3), in which case $\nu = 1$. In that case, the natural confidence regions in the ν-dimensional subspace of the M-dimensional parameter space are the *projections* of the M-dimensional regions defined by fixed $\Delta\chi^2$ into the ν-dimensional spaces of interest. In Figure 15.6.4, for the case $M = 2$, we show regions corresponding to several values of $\Delta\chi^2$. The one-dimensional confidence interval in a_2 corresponding to the region bounded by $\Delta\chi^2 = 1$ lies between the lines A and A'.

Notice that the projection of the higher-dimensional region on the lower-dimension space is used, not the intersection. The intersection would be the band between Z and Z'. It is *never* used. It is shown in the figure only for the purpose of

making this cautionary point, that it should not be confused with the projection.

Probability Distribution of Parameters in the Normal Case

You may be wondering why we have, in this section up to now, made no connection at all with the error estimates that come out of the χ^2 fitting procedure, most notably the covariance matrix C_{ij}. The reason is this: χ^2 minimization is a useful means for estimating parameters even if the measurement errors are not normally distributed. While normally distributed errors are required if the χ^2 parameter estimate is to be a maximum likelihood estimator (§15.1), one is often willing to give up that property in return for the relative convenience of the χ^2 procedure. Only in extreme cases, measurement error distributions with very large "tails," is χ^2 minimization abandoned in favor of more robust techniques, as will be discussed in §15.7.

However, the formal covariance matrix that comes out of a χ^2 minimization has a clear quantitative interpretation only if (or to the extent that) the measurement errors actually are normally distributed. In the case of *non*normal errors, you are "allowed"

- to fit for parameters by minimizing χ^2
- to use a contour of constant $\Delta\chi^2$ as the boundary of your confidence region
- to use Monte Carlo simulation or detailed analytic calculation in determining *which* contour $\Delta\chi^2$ is the correct one for your desired confidence level
- to give the covariance matrix C_{ij} as the "formal covariance matrix of the fit."

You are *not* allowed

- to use formulas that we now give for the case of normal errors, which establish quantitative relationships among $\Delta\chi^2$, C_{ij}, and the confidence level.

Here are the key theorems that hold when (i) the measurement errors are normally distributed, and either (ii) the model is linear in its parameters or (iii) the sample size is large enough that the uncertainties in the fitted parameters **a** do not extend outside a region in which the model could be replaced by a suitable linearized model. [Note that condition (iii) does not preclude your use of a nonlinear routine like `mqrfit` to *find* the fitted parameters.]

Theorem A. $\chi^2_{\min}$ is distributed as a chi-square distribution with $N - M$ degrees of freedom, where N is the number of data points and M is the number of fitted parameters. This is the basic theorem that lets you evaluate the goodness-of-fit of the model, as discussed above in §15.1. We list it first to remind you that unless the goodness-of-fit is credible, the whole estimation of parameters is suspect.

Theorem B. If $\mathbf{a}^S_{(j)}$ is drawn from the universe of simulated data sets with actual parameters $\mathbf{a}_{(0)}$, then the probability distribution of $\delta\mathbf{a} \equiv \mathbf{a}^S_{(j)} - \mathbf{a}_{(0)}$ is the multivariate normal distribution

$$P(\delta\mathbf{a})\, da_1 \ldots da_M = \text{const.} \times \exp\left(-\frac{1}{2}\delta\mathbf{a} \cdot [\alpha] \cdot \delta\mathbf{a}\right)\, da_1 \ldots da_M$$

where $[\alpha]$ is the curvature matrix defined in equation (15.5.8).

Theorem C. If $\mathbf{a}^S_{(j)}$ is drawn from the universe of simulated data sets with actual parameters $\mathbf{a}_{(0)}$, then the quantity $\Delta\chi^2 \equiv \chi^2(\mathbf{a}_{(j)}) - \chi^2(\mathbf{a}_{(0)})$ is distributed

as a chi-square distribution with M degrees of freedom. Here the χ^2's are all evaluated using the fixed (actual) data set $\mathcal{D}_{(0)}$. This theorem makes the connection between particular values of $\Delta\chi^2$ and the fraction of the probability distribution that they enclose as an M-dimensional region, i.e., the confidence level of the M-dimensional confidence region.

Theorem D. Suppose that $\mathbf{a}^S_{(j)}$ is drawn from the universe of simulated data sets (as above), that its first ν components $a_1, \ldots, a_\nu$ are held fixed, and that its remaining $M - \nu$ components are varied so as to minimize χ^2. Call this minimum value χ^2_ν. Then $\Delta\chi^2_\nu \equiv \chi^2_\nu - \chi^2_{\min}$ is distributed as a chi-square distribution with ν degrees of freedom. If you consult Figure 15.6.4, you will see that this theorem connects the *projected* $\Delta\chi^2$ region with a confidence level. In the figure, a point that is held fixed in a_2 and allowed to vary in a_1 minimizing χ^2 will seek out the ellipse whose top or bottom edge is tangent to the line of constant a_2, and is therefore the line that projects it onto the smaller-dimensional space.

As a first example, let us consider the case $\nu = 1$, where we want to find the confidence interval of a single parameter, say a_1. Notice that the chi-square distribution with $\nu = 1$ degree of freedom is the same distribution as that of the square of a single normally distributed quantity. Thus $\Delta\chi^2_\nu < 1$ occurs 68.3 percent of the time (1-σ for the normal distribution), $\Delta\chi^2_\nu < 4$ occurs 95.4 percent of the time (2-σ for the normal distribution), $\Delta\chi^2_\nu < 9$ occurs 99.73 percent of the time (3-σ for the normal distribution), etc. In this manner you find the $\Delta\chi^2_\nu$ that corresponds to your desired confidence level. (Additional values are given in the accompanying table.)

Let $\delta\mathbf{a}$ be a change in the parameters whose first component is arbitrary, δa_1, but the rest of whose components are chosen to minimize the $\Delta\chi^2$. Then Theorem D applies. The value of $\Delta\chi^2$ is given in general by

$$\Delta\chi^2 = \delta\mathbf{a} \cdot [\alpha] \cdot \delta\mathbf{a} \tag{15.6.1}$$

which follows from equation (15.5.8) applied at $\chi^2_{\min}$ where $\beta_k = 0$. Since $\delta\mathbf{a}$ by hypothesis minimizes χ^2 in all but its first component, the second through Mth components of the normal equations (15.5.9) continue to hold. Therefore, the solution of (15.5.9) is

$$\delta\mathbf{a} = [\alpha]^{-1} \cdot \begin{pmatrix} c \\ 0 \\ \vdots \\ 0 \end{pmatrix} = [C] \cdot \begin{pmatrix} c \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{15.6.2}$$

where c is one arbitrary constant that we get to adjust to make (15.6.1) give the desired left-hand value. Plugging (15.6.2) into (15.6.1) and using the fact that $[C]$ and $[\alpha]$ are inverse matrices of one another, we get

$$c = \delta a_1 / C_{11} \qquad \text{and} \qquad \Delta\chi^2_\nu = (\delta a_1)^2 / C_{11} \tag{15.6.3}$$

or

$$\delta a_1 = \pm\sqrt{\Delta\chi^2_\nu}\,\sqrt{C_{11}} \tag{15.6.4}$$

At last! A relation between the confidence interval $\pm\delta a_1$ and the formal standard error $\sigma_1 \equiv \sqrt{C_{11}}$. Not unreasonably, we find that the 68 percent confidence interval is $\pm\sigma_1$, the 95 percent confidence interval is $\pm 2\sigma_1$, etc.

$\Delta\chi^2$ as a Function of Confidence Level and Degrees of Freedom						
	ν					
p	1	2	3	4	5	6
68.3%	1.00	2.30	3.53	4.72	5.89	7.04
90%	2.71	4.61	6.25	7.78	9.24	10.6
95.4%	4.00	6.17	8.02	9.70	11.3	12.8
99%	6.63	9.21	11.3	13.3	15.1	16.8
99.73%	9.00	11.8	14.2	16.3	18.2	20.1
99.99%	15.1	18.4	21.1	23.5	25.7	27.8

These considerations hold not just for the individual parameters a_i, but also for any linear combination of them: If

$$b \equiv \sum_{k=1}^{M} c_i a_i = \mathbf{c} \cdot \mathbf{a} \tag{15.6.5}$$

then the 68 percent confidence interval on b is

$$\delta b = \pm\sqrt{\mathbf{c} \cdot [C] \cdot \mathbf{c}} \tag{15.6.6}$$

However, these simple, normal-sounding numerical relationships do *not* hold in the case $\nu > 1$ [3]. In particular, $\Delta\chi^2 = 1$ is not the boundary, nor does it project onto the boundary, of a 68.3 percent confidence region when $\nu > 1$. If you want to calculate not confidence intervals in one parameter, but confidence ellipses in two parameters jointly, or ellipsoids in three, or higher, then you must follow the following prescription for implementing Theorems C and D above:

- Let ν be the number of fitted parameters whose joint confidence region you wish to display, $\nu \leq M$. Call these parameters the "parameters of interest."
- Let p be the confidence limit desired, e.g., $p = 0.68$ or $p = 0.95$.
- Find Δ (i.e., $\Delta\chi^2$) such that the probability of a chi-square variable with ν degrees of freedom being less than Δ is p. For some useful values of p and ν, Δ is given in the table. For other values, you can use the routine `gammq` and a simple root-finding routine (e.g., bisection) to find Δ such that $\texttt{gammq}(\nu/2, \Delta/2) = 1 - p$.
- Take the $M \times M$ covariance matrix $[C] = [\alpha]^{-1}$ of the chi-square fit. Copy the intersection of the ν rows and columns corresponding to the parameters of interest into a $\nu \times \nu$ matrix denoted $[C_{\rm proj}]$.
- Invert the matrix $[C_{\rm proj}]$. (In the one-dimensional case this was just taking the reciprocal of the element C_{11}.)
- The equation for the elliptical boundary of your desired confidence region in the ν-dimensional subspace of interest is

$$\Delta = \delta\mathbf{a}' \cdot [C_{\rm proj}]^{-1} \cdot \delta\mathbf{a}' \tag{15.6.7}$$

  where $\delta\mathbf{a}'$ is the ν-dimensional vector of parameters of interest.

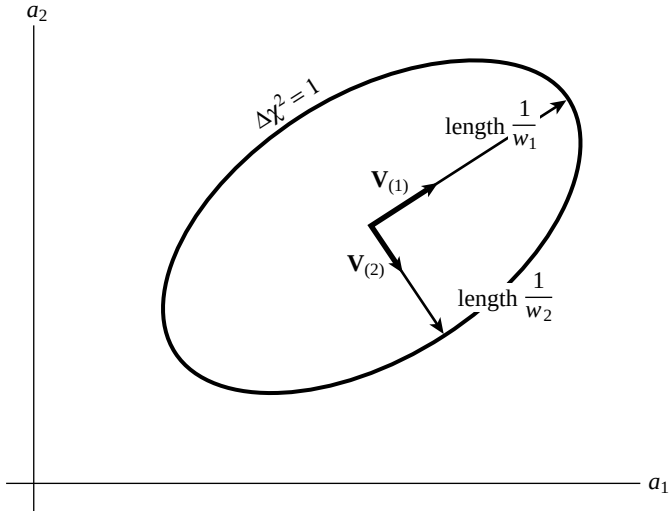


Figure 15.6.5. Relation of the confidence region ellipse $\Delta\chi^2 = 1$ to quantities computed by singular value decomposition. The vectors $\mathbf{V}_{(i)}$ are unit vectors along the principal axes of the confidence region. The semi-axes have lengths equal to the reciprocal of the singular values w_i. If the axes are all scaled by some constant factor α, $\Delta\chi^2$ is scaled by the factor α^2.

If you are confused at this point, you may find it helpful to compare Figure 15.6.4 and the accompanying table, considering the case $M = 2$ with $\nu = 1$ and $\nu = 2$. You should be able to verify the following statements: (i) The horizontal band between C and C' contains 99 percent of the probability distribution, so it is a confidence limit on a_2 alone at this level of confidence. (ii) Ditto the band between B and B' at the 90 percent confidence level. (iii) The dashed ellipse, labeled by $\Delta\chi^2 = 2.30$, contains 68.3 percent of the probability distribution, so it is a confidence region for a_1 and a_2 jointly, at this level of confidence.

Confidence Limits from Singular Value Decomposition

When you have obtained your χ^2 fit by singular value decomposition (§15.4), the information about the fit's formal errors comes packaged in a somewhat different, but generally more convenient, form. The columns of the matrix $\mathbf{V}$ are an orthonormal set of M vectors that are the principal axes of the $\Delta\chi^2 =$ constant ellipsoids. We denote the columns as $\mathbf{V}_{(1)} \ldots \mathbf{V}_{(M)}$. The lengths of those axes are inversely proportional to the corresponding singular values $w_1 \ldots w_M$; see Figure 15.6.5. The boundaries of the ellipsoids are thus given by

$$\Delta\chi^2 = w_1^2(\mathbf{V}_{(1)} \cdot \delta\mathbf{a})^2 + \cdots + w_M^2(\mathbf{V}_{(M)} \cdot \delta\mathbf{a})^2 \tag{15.6.8}$$

which is the justification for writing equation (15.4.18) above. Keep in mind that it is *much* easier to plot an ellipsoid given a list of its vector principal axes, than given its matrix quadratic form!

The formula for the covariance matrix $[C]$ in terms of the columns $\mathbf{V}_{(i)}$ is

$$[C] = \sum_{i=1}^{M} \frac{1}{w_i^2} \mathbf{V}_{(i)} \otimes \mathbf{V}_{(i)} \tag{15.6.9}$$

or, in components,

$$C_{jk} = \sum_{i=1}^{M} \frac{1}{w_i^2} V_{ji} V_{ki} \tag{15.6.10}$$

CITED REFERENCES AND FURTHER READING:

Efron, B. 1982, *The Jackknife, the Bootstrap, and Other Resampling Plans* (Philadelphia: S.I.A.M.). [1]

Efron, B., and Tibshirani, R. 1986, *Statistical Science* vol. 1, pp. 54–77. [2]

Avni, Y. 1976, *Astrophysical Journal*, vol. 210, pp. 642–646. [3]

Lampton, M., Margon, M., and Bowyer, S. 1976, *Astrophysical Journal*, vol. 208, pp. 177–190.

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

15.7 Robust Estimation

The concept of *robustness* has been mentioned in passing several times already. In §14.1 we noted that the median was a more robust estimator of central value than the mean; in §14.6 it was mentioned that rank correlation is more robust than linear correlation. The concept of outlier points as exceptions to a Gaussian model for experimental error was discussed in §15.1.

The term "robust" was coined in statistics by G.E.P. Box in 1953. Various definitions of greater or lesser mathematical rigor are possible for the term, but in general, referring to a statistical estimator, it means "insensitive to small departures from the idealized assumptions for which the estimator is optimized." [1,2] The word "small" can have two different interpretations, both important: either fractionally small departures for all data points, or else fractionally large departures for a small number of data points. It is the latter interpretation, leading to the notion of outlier points, that is generally the most stressful for statistical procedures.

Statisticians have developed various sorts of robust statistical estimators. Many, if not most, can be grouped in one of three categories.

M-estimates follow from maximum-likelihood arguments very much as equations (15.1.5) and (15.1.7) followed from equation (15.1.3). M-estimates are usually the most relevant class for model-fitting, that is, estimation of parameters. We therefore consider these estimates in some detail below.

L-estimates are "linear combinations of order statistics." These are most applicable to estimations of central value and central tendency, though they can occasionally be applied to some problems in estimation of parameters. Two "typical" L-estimates will give you the general idea. They are (i) the median, and (ii) *Tukey's trimean*, defined as the weighted average of the first, second, and third quartile points in a distribution, with weights 1/4, 1/2, and 1/4, respectively.

R-estimates are estimates based on rank tests. For example, the equality or inequality of two distributions can be estimated by the *Wilcoxon test* of computing the mean rank of one distribution in a combined sample of both distributions. The Kolmogorov-Smirnov statistic (equation 14.3.6) and the Spearman rank-order

$$C_{jk} = \sum_{i=1}^{M} \frac{1}{w_i^2} V_{ji} V_{ki} \tag{15.6.10}$$

CITED REFERENCES AND FURTHER READING:

Efron, B. 1982, *The Jackknife, the Bootstrap, and Other Resampling Plans* (Philadelphia: S.I.A.M.). [1]

Efron, B., and Tibshirani, R. 1986, *Statistical Science* vol. 1, pp. 54–77. [2]

Avni, Y. 1976, *Astrophysical Journal*, vol. 210, pp. 642–646. [3]

Lampton, M., Margon, M., and Bowyer, S. 1976, *Astrophysical Journal*, vol. 208, pp. 177–190.

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

15.7 Robust Estimation

The concept of *robustness* has been mentioned in passing several times already. In §14.1 we noted that the median was a more robust estimator of central value than the mean; in §14.6 it was mentioned that rank correlation is more robust than linear correlation. The concept of outlier points as exceptions to a Gaussian model for experimental error was discussed in §15.1.

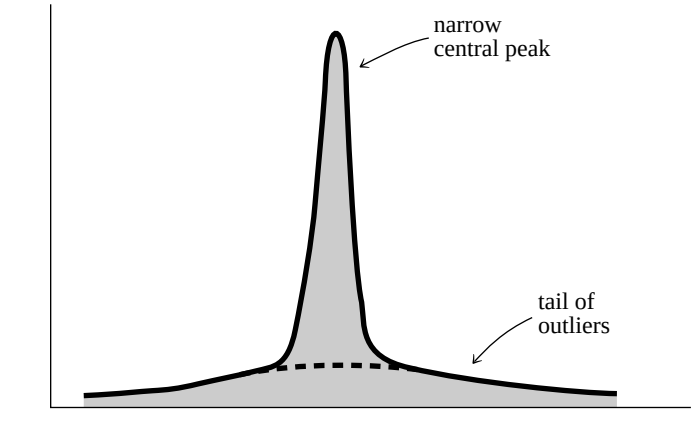
The term "robust" was coined in statistics by G.E.P. Box in 1953. Various definitions of greater or lesser mathematical rigor are possible for the term, but in general, referring to a statistical estimator, it means "insensitive to small departures from the idealized assumptions for which the estimator is optimized." [1,2] The word "small" can have two different interpretations, both important: either fractionally small departures for all data points, or else fractionally large departures for a small number of data points. It is the latter interpretation, leading to the notion of outlier points, that is generally the most stressful for statistical procedures.

Statisticians have developed various sorts of robust statistical estimators. Many, if not most, can be grouped in one of three categories.

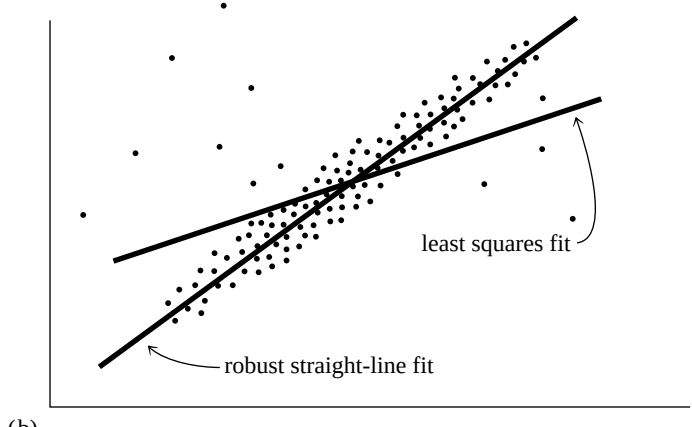
M-estimates follow from maximum-likelihood arguments very much as equations (15.1.5) and (15.1.7) followed from equation (15.1.3). M-estimates are usually the most relevant class for model-fitting, that is, estimation of parameters. We therefore consider these estimates in some detail below.

L-estimates are "linear combinations of order statistics." These are most applicable to estimations of central value and central tendency, though they can occasionally be applied to some problems in estimation of parameters. Two "typical" L-estimates will give you the general idea. They are (i) the median, and (ii) *Tukey's trimean*, defined as the weighted average of the first, second, and third quartile points in a distribution, with weights 1/4, 1/2, and 1/4, respectively.

R-estimates are estimates based on rank tests. For example, the equality or inequality of two distributions can be estimated by the *Wilcoxon test* of computing the mean rank of one distribution in a combined sample of both distributions. The Kolmogorov-Smirnov statistic (equation 14.3.6) and the Spearman rank-order



(a)



(b)

Figure 15.7.1. Examples where robust statistical methods are desirable: (a) A one-dimensional distribution with a tail of outliers; statistical fluctuations in these outliers can prevent accurate determination of the position of the central peak. (b) A distribution in two dimensions fitted to a straight line; non-robust techniques such as least-squares fitting can have undesired sensitivity to outlying points.

correlation coefficient (14.6.1) are R-estimates in essence, if not always by formal definition.

Some other kinds of robust techniques, coming from the fields of optimal control and filtering rather than from the field of mathematical statistics, are mentioned at the end of this section. Some examples where robust statistical methods are desirable are shown in Figure 15.7.1.

Estimation of Parameters by Local M-Estimates

Suppose we know that our measurement errors are not normally distributed. Then, in deriving a maximum-likelihood formula for the estimated parameters **a** in a model $y(x; \mathbf{a})$, we would write instead of equation (15.1.3)

$$P = \prod_{i=1}^{N} \{\exp\left[-\rho(y_i, y\{x_i; \mathbf{a}\})\right] \Delta y\} \tag{15.7.1}$$

where the function ρ is the negative logarithm of the probability density. Taking the logarithm of (15.7.1) analogously with (15.1.4), we find that we want to minimize the expression

$$\sum_{i=1}^{N} \rho(y_i, y\{x_i; \mathbf{a}\}) \tag{15.7.2}$$

Very often, it is the case that the function ρ depends not independently on its two arguments, measured y_i and predicted $y(x_i)$, but only on their difference, at least if scaled by some weight factors σ_i which we are able to assign to each point. In this case the M-estimate is said to be *local*, and we can replace (15.7.2) by the prescription

$$\text{minimize over } \mathbf{a} \qquad \sum_{i=1}^{N} \rho\left(\frac{y_i - y(x_i; \mathbf{a})}{\sigma_i}\right) \tag{15.7.3}$$

where the function $\rho(z)$ is a function of a single variable $z \equiv [y_i - y(x_i)]/\sigma_i$.

If we now define the derivative of $\rho(z)$ to be a function $\psi(z)$,

$$\psi(z) \equiv \frac{d\rho(z)}{dz} \tag{15.7.4}$$

then the generalization of (15.1.7) to the case of a general M-estimate is

$$0 = \sum_{i=1}^{N} \frac{1}{\sigma_i} \psi\left(\frac{y_i - y(x_i)}{\sigma_i}\right)\left(\frac{\partial y(x_i; \mathbf{a})}{\partial a_k}\right) \qquad k = 1, \ldots, M \tag{15.7.5}$$

If you compare (15.7.3) to (15.1.3), and (15.7.5) to (15.1.7), you see at once that the specialization for normally distributed errors is

$$\rho(z) = \frac{1}{2}z^2 \qquad \psi(z) = z \qquad \text{(normal)} \tag{15.7.6}$$

If the errors are distributed as a *double* or *two-sided exponential*, namely

$$\text{Prob}\,\{y_i - y(x_i)\} \sim \exp\left(-\left|\frac{y_i - y(x_i)}{\sigma_i}\right|\right) \tag{15.7.7}$$

then, by contrast,

$$\rho(x) = |z| \qquad \psi(z) = \text{sgn}(z) \qquad \text{(double exponential)} \tag{15.7.8}$$

Comparing to equation (15.7.3), we see that in this case the maximum likelihood estimator is obtained by minimizing the *mean absolute deviation*, rather than the mean square deviation. Here the tails of the distribution, although exponentially decreasing, are asymptotically much larger than any corresponding Gaussian.

A distribution with even more extensive — therefore sometimes even more realistic — tails is the *Cauchy* or *Lorentzian* distribution,

$$\text{Prob}\,\{y_i - y(x_i)\} \sim \frac{1}{1 + \frac{1}{2}\left(\frac{y_i - y(x_i)}{\sigma_i}\right)^2} \tag{15.7.9}$$

This implies

$$\rho(z) = \log\left(1 + \frac{1}{2}z^2\right) \qquad \psi(z) = \frac{z}{1 + \frac{1}{2}z^2} \qquad \text{(Lorentzian)} \qquad (15.7.10)$$

Notice that the ψ function occurs as a weighting function in the generalized normal equations (15.7.5). For normally distributed errors, equation (15.7.6) says that the more deviant the points, the greater the weight. By contrast, when tails are somewhat more prominent, as in (15.7.7), then (15.7.8) says that all deviant points get the same relative weight, with only the sign information used. Finally, when the tails are even larger, (15.7.10) says the ψ increases with deviation, then starts *decreasing*, so that very deviant points — the true outliers — are not counted at all in the estimation of the parameters.

This general idea, that the weight given individual points should first increase with deviation, then decrease, motivates some additional prescriptions for ψ which do not especially correspond to standard, textbook probability distributions. Two examples are

Andrew's sine

$$\psi(z) = \begin{cases} \sin(z/c) & |z| < c\pi \\ 0 & |z| > c\pi \end{cases} \qquad (15.7.11)$$

If the measurement errors happen to be normal after all, with standard deviations σ_i, then it can be shown that the optimal value for the constant c is $c = 2.1$.

Tukey's biweight

$$\psi(z) = \begin{cases} z(1 - z^2/c^2)^2 & |z| < c \\ 0 & |z| > c \end{cases} \qquad (15.7.12)$$

where the optimal value of c for normal errors is $c = 6.0$.

Numerical Calculation of M-Estimates

To fit a model by means of an M-estimate, you first decide which M-estimate you want, that is, which matching pair ρ, ψ you want to use. We rather like (15.7.8) or (15.7.10).

You then have to make an unpleasant choice between two fairly difficult problems. Either find the solution of the nonlinear set of M equations (15.7.5), or else minimize the single function in M variables (15.7.3).

Notice that the function (15.7.8) has a discontinuous ψ, and a discontinuous derivative for ρ. Such discontinuities frequently wreak havoc on both general nonlinear equation solvers and general function minimizing routines. You might now think of rejecting (15.7.8) in favor of (15.7.10), which is smoother. However, you will find that the latter choice is also bad news for many general equation solving or minimization routines: small changes in the fitted parameters can drive $\psi(z)$ off its peak into one or the other of its asymptotically small regimes. Therefore, different terms in the equation spring into or out of action (almost as bad as analytic discontinuities).

Don't despair. If your computer budget (or, for personal computers, patience) is up to it, this is an excellent application for the downhill simplex minimization

algorithm exemplified in `amoeba` §10.4 or `amebsa` in §10.9. Those algorithms make no assumptions about continuity; they just ooze downhill and will work for virtually any sane choice of the function ρ.

It is very much to your (financial) advantage to find good starting values, however. Often this is done by first fitting the model by the standard χ^2 (nonrobust) techniques, e.g., as described in §15.4 or §15.5. The fitted parameters thus obtained are then used as starting values in `amoeba`, now using the robust choice of ρ and minimizing the expression (15.7.3).

Fitting a Line by Minimizing Absolute Deviation

Occasionally there is a special case that happens to be much easier than is suggested by the general strategy outlined above. The case of equations (15.7.7)–(15.7.8), when the model is a simple straight line

$$y(x; a, b) = a + bx \tag{15.7.13}$$

and where the weights σ_i are all equal, happens to be such a case. The problem is precisely the robust version of the problem posed in equation (15.2.1) above, namely fit a straight line through a set of data points. The merit function to be minimized is

$$\sum_{i=1}^{N} |y_i - a - bx_i| \tag{15.7.14}$$

rather than the χ^2 given by equation (15.2.2).

The key simplification is based on the following fact: The median c_M of a set of numbers c_i is also that value which minimizes the sum of the absolute deviations

$$\sum_i |c_i - c_M|$$

(Proof: Differentiate the above expression with respect to c_M and set it to zero.)

It follows that, for fixed b, the value of a that minimizes (15.7.14) is

$$a = \text{median}\,\{y_i - bx_i\} \tag{15.7.15}$$

Equation (15.7.5) for the parameter b is

$$0 = \sum_{i=1}^{N} x_i\ \text{sgn}(y_i - a - bx_i) \tag{15.7.16}$$

(where $\text{sgn}(0)$ is to be interpreted as zero). If we replace a in this equation by the implied function $a(b)$ of (15.7.15), then we are left with an equation in a single variable which can be solved by bracketing and bisection, as described in §9.1. (In fact, it is dangerous to use any fancier method of root-finding, because of the discontinuities in equation 15.7.16.)

Here is a routine that does all this. It calls `select` (§8.5) to find the median. The bracketing and bisection are built in to the routine, as is the χ^2 solution that generates the initial guesses for a and b. Notice that the evaluation of the right-hand side of (15.7.16) occurs in the function `rofunc`, with communication via a common block. To save memory, you could generate your data arrays directly into that common block, deleting them from this routine's calling sequence.

```
      SUBROUTINE medfit(x,y,ndata,a,b,abdev)
      INTEGER ndata,NMAX,ndatat
      PARAMETER (NMAX=1000)
      REAL a,abdev,b,x(ndata),y(ndata),
     *     arr(NMAX),xt(NMAX),yt(NMAX),aa,abdevt
      COMMON /arrays/ xt,yt,arr,aa,abdevt,ndatat
C     USES rofunc
```

Fits $y = a + bx$ by the criterion of least absolute deviations. The arrays `x(1:ndata)` and `y(1:ndata)` are the input experimental points. The fitted parameters `a` and `b` are output, along with `abdev`, which is the mean absolute deviation (in y) of the experimental points from the fitted line. This routine uses the routine `rofunc`, with communication via a common block.

```
      INTEGER j
      REAL b1,b2,bb,chisq,del,f,f1,f2,sigb,sx,sxx,sxy,sy,rofunc
      sx=0.
      sy=0.
      sxy=0.
      sxx=0.
      do 11 j=1,ndata                 As a first guess for a and b, we will find the least-squares
        xt(j)=x(j)                      fitting line.
        yt(j)=y(j)
        sx=sx+x(j)
        sy=sy+y(j)
        sxy=sxy+x(j)*y(j)
        sxx=sxx+x(j)**2
      enddo 11
      ndatat=ndata
      del=ndata*sxx-sx**2
      aa=(sxx*sy-sx*sxy)/del          Least-squares solutions.
      bb=(ndata*sxy-sx*sy)/del
      chisq=0.
      do 12 j=1,ndata
        chisq=chisq+(y(j)-(aa+bb*x(j)))**2
      enddo 12
      sigb=sqrt(chisq/del)            The standard deviation will give some idea of how big an
      b1=bb                             iteration step to take.
      f1=rofunc(b1)
      b2=bb+sign(3.*sigb,f1)          Guess bracket as 3-$\sigma$ away, in the downhill direction known
      f2=rofunc(b2)                     from f1.
      if(b2.eq.b1)then
        a=aa
        b=bb
        abdev=abdevt/ndata
        return
      endif
1     if(f1*f2.gt.0.)then             Bracketing.
        bb=b2+1.6*(b2-b1)
        b1=b2
        f1=f2
        b2=bb
        f2=rofunc(b2)
        goto 1
      endif
      sigb=0.01*sigb                  Refine until error a negligible number of standard deviations.
2     if(abs(b2-b1).gt.sigb)then      Bisection.
        bb=b1+0.5*(b2-b1)
        if(bb.eq.b1.or.bb.eq.b2)goto 3
        f=rofunc(bb)
        if(f*f1.ge.0.)then
          f1=f
          b1=bb
        else
          f2=f
          b2=bb
        endif
        goto 2
      endif
```

```
      b=bb
      abdev=abdevt/ndata
      return
      END


      FUNCTION rofunc(b)
      INTEGER NMAX
      REAL rofunc,b,EPS
      PARAMETER (NMAX=1000,EPS=1.e-7)
C     USES select
        Evaluates the right-hand side of equation (15.7.16) for a given value of b. Communication
        with the program medfit is through a common block.
      INTEGER j,ndata
      REAL aa,abdev,d,sum,arr(NMAX),x(NMAX),y(NMAX),select
      COMMON /arrays/ x,y,arr,aa,abdev,ndata
      do 11 j=1,ndata
          arr(j)=y(j)-b*x(j)
      enddo 11
      if (mod(ndata,2).eq.0) then
          j=ndata/2
          aa=0.5*(select(j,ndata,arr)+select(j+1,ndata,arr))
      else
          aa=select((ndata+1)/2,ndata,arr)
      endif
      sum=0.
      abdev=0.
      do 12 j=1,ndata
          d=y(j)-(b*x(j)+aa)
          abdev=abdev+abs(d)
          if (y(j).ne.0.) d=d/abs(y(j))
          if (abs(d).gt.EPS) sum=sum+x(j)*sign(1.0,d)
      enddo 12
      rofunc=sum
      return
      END
```

Other Robust Techniques

Sometimes you may have *a priori* knowledge about the probable values and probable uncertainties of some parameters that you are trying to estimate from a data set. In such cases you may want to perform a fit that takes this advance information properly into account, neither completely freezing a parameter at a predetermined value (as in `lfit` §15.4) nor completely leaving it to be determined by the data set. The formalism for doing this is called "use of *a priori* covariances."

A related problem occurs in signal processing and control theory, where it is sometimes desired to "track" (i.e., maintain an estimate of) a time-varying signal in the presence of noise. If the signal is known to be characterized by some number of parameters that vary only slowly, then the formalism of *Kalman filtering* tells how the incoming, raw measurements of the signal should be processed to produce best parameter estimates as a function of time. For example, if the signal is a frequency-modulated sine wave, then the slowly varying parameter might be the instantaneous frequency. The Kalman filter for this case is called a *phase-locked loop* and is implemented in the circuitry of good radio receivers [3,4].

CITED REFERENCES AND FURTHER READING:

Huber, P.J. 1981, *Robust Statistics* (New York: Wiley). [1]

Launer, R.L., and Wilkinson, G.N. (eds.) 1979, *Robustness in Statistics* (New York: Academic Press). [2]

Bryson, A. E., and Ho, Y.C. 1969, *Applied Optimal Control* (Waltham, MA: Ginn). [3]

Jazwinski, A. H. 1970, *Stochastic Processes and Filtering Theory* (New York: Academic Press). [4]

Chapter 16. Integration of Ordinary Differential Equations

16.0 Introduction

Problems involving ordinary differential equations (ODEs) can always be reduced to the study of sets of first-order differential equations. For example the second-order equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{16.0.1}$$

can be rewritten as two first-order equations

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned} \tag{16.0.2}$$

where z is a new variable. This exemplifies the procedure for an arbitrary ODE. The usual choice for the new variables is to let them be just derivatives of each other (and of the original variable). Occasionally, it is useful to incorporate into their definition some other factors in the equation, or some powers of the independent variable, for the purpose of mitigating singular behavior that could result in overflows or increased roundoff error. Let common sense be your guide: If you find that the original variables are smooth in a solution, while your auxiliary variables are doing crazy things, then figure out why and choose different auxiliary variables.

The generic problem in ordinary differential equations is thus reduced to the study of a set of N coupled *first-order* differential equations for the functions $y_i,\ i = 1, 2, \ldots, N$, having the general form

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \qquad i = 1, \ldots, N \tag{16.0.3}$$

where the functions f_i on the right-hand side are known.

A problem involving ODEs is not completely specified by its equations. Even more crucial in determining how to attack the problem numerically is the nature of the problem's boundary conditions. Boundary conditions are algebraic conditions

on the values of the functions y_i in (16.0.3). In general they can be satisfied at discrete specified points, but do not hold between those points, i.e., are not preserved automatically by the differential equations. Boundary conditions can be as simple as requiring that certain variables have certain numerical values, or as complicated as a set of nonlinear algebraic equations among the variables.

Usually, it is the nature of the boundary conditions that determines which numerical methods will be feasible. Boundary conditions divide into two broad categories.

- In *initial value problems* all the y_i are given at some starting value x_s, and it is desired to find the y_i's at some final point x_f, or at some discrete list of points (for example, at tabulated intervals).
- In *two-point boundary value problems*, on the other hand, boundary conditions are specified at more than one x. Typically, some of the conditions will be specified at x_s and the remainder at x_f.

This chapter will consider exclusively the initial value problem, deferring two-point boundary value problems, which are generally more difficult, to Chapter 17.

The underlying idea of any routine for solving the initial value problem is always this: Rewrite the dy's and dx's in (16.0.3) as finite steps Δy and Δx, and multiply the equations by Δx. This gives algebraic formulas for the change in the functions when the independent variable x is "stepped" by one "stepsize" Δx. In the limit of making the stepsize very small, a good approximation to the underlying differential equation is achieved. Literal implementation of this procedure results in *Euler's method* (16.1.1, below), which is, however, *not* recommended for any practical use. Euler's method is conceptually important, however; one way or another, practical methods all come down to this same idea: Add small increments to your functions corresponding to derivatives (right-hand sides of the equations) multiplied by stepsizes.

In this chapter we consider three major types of practical numerical methods for solving initial value problems for ODEs:

- Runge-Kutta methods
- Richardson extrapolation and its particular implementation as the Bulirsch-Stoer method
- predictor-corrector methods.

A brief description of each of these types follows.

1. *Runge-Kutta* methods propagate a solution over an interval by combining the information from several Euler-style steps (each involving one evaluation of the right-hand f's), and then using the information obtained to match a Taylor series expansion up to some higher order.

2. *Richardson extrapolation* uses the powerful idea of extrapolating a computed result to the value that *would* have been obtained if the stepsize had been very much smaller than it actually was. In particular, extrapolation to zero stepsize is the desired goal. The first practical ODE integrator that implemented this idea was developed by Bulirsch and Stoer, and so extrapolation methods are often called Bulirsch-Stoer methods.

3. *Predictor-corrector* methods store the solution along the way, and use those results to extrapolate the solution one step advanced; they then correct the extrapolation using derivative information at the new point. These are best for very smooth functions.

Runge-Kutta is what you use when (i) you don't know any better, or (ii) you have an intransigent problem where Bulirsch-Stoer is failing, or (iii) you have a trivial problem where computational efficiency is of no concern. Runge-Kutta succeeds virtually always; but it is not usually fastest, except when evaluating f_i is cheap and moderate accuracy ($\lesssim 10^{-5}$) is required. Predictor-corrector methods, since they use past information, are somewhat more difficult to start up, but, for many smooth problems, they are computationally more efficient than Runge-Kutta. In recent years Bulirsch-Stoer has been replacing predictor-corrector in many applications, but it is too soon to say that predictor-corrector is dominated in all cases. However, it appears that only rather sophisticated predictor-corrector routines are competitive. Accordingly, we have chosen *not* to give an implementation of predictor-corrector in this book. We discuss predictor-corrector further in §16.7, so that you can use a canned routine should you encounter a suitable problem. In our experience, the relatively simple Runge-Kutta and Bulirsch-Stoer routines we give are adequate for most problems.

Each of the three types of methods can be organized to monitor internal consistency. This allows numerical errors which are inevitably introduced into the solution to be controlled by automatic, (*adaptive*) changing of the fundamental stepsize. We always recommend that adaptive stepsize control be implemented, and we will do so below.

In general, all three types of methods can be applied to any initial value problem. Each comes with its own set of debits and credits that must be understood before it is used.

We have organized the routines in this chapter into three nested levels. The lowest or "nitty-gritty" level is the piece we call the *algorithm* routine. This implements the basic formulas of the method, starts with dependent variables y_i at x, and returns new values of the dependent variables at the value $x + h$. The algorithm routine also yields up some information about the quality of the solution after the step. The routine is dumb, however, and it is unable to make any adaptive decision about whether the solution is of acceptable quality or not.

That quality-control decision we encode in a *stepper* routine. The stepper routine calls the algorithm routine. It may reject the result, set a smaller stepsize, and call the algorithm routine again, until compatibility with a predetermined accuracy criterion has been achieved. The stepper's fundamental task is to take the largest stepsize consistent with specified performance. Only when this is accomplished does the true power of an algorithm come to light.

Above the stepper is the *driver* routine, which starts and stops the integration, stores intermediate results, and generally acts as an interface with the user. There is nothing at all canonical about our driver routines. You should consider them to be examples, and you can customize them for your particular application.

Of the routines that follow, `rk4`, `rkck`, `mmid`, `stoerm`, and `simpr` are algorithm routines; `rkqs`, `bsstep`, `stiff`, and `stifbs` are steppers; `rkdumb` and `odeint` are drivers.

Section 16.6 of this chapter treats the subject of *stiff equations*, relevant both to ordinary differential equations and also to partial differential equations (Chapter 19).

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.

Lambert, J. 1973, *Computational Methods in Ordinary Differential Equations* (New York: Wiley).

Lapidus, L., and Seinfeld, J. 1971, *Numerical Solution of Ordinary Differential Equations* (New York: Academic Press).

16.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{16.1.1}$$

which advances a solution from x_n to $x_{n+1} \equiv x_n + h$. The formula is unsymmetrical: It advances the solution through an interval h, but uses derivative information only at the beginning of that interval (see Figure 16.1.1). That means (and you can verify by expansion in power series) that the step's error is only one power of h smaller than the correction, i.e $O(h^2)$ added to (16.1.1).

There are several reasons that Euler's method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other, fancier, methods run at the equivalent stepsize, and (ii) neither is it very stable (see §16.6 below).

Consider, however, the use of a step like (16.1.1) to take a "trial" step to the midpoint of the interval. Then use the value of both x and y at that midpoint to compute the "real" step across the whole interval. Figure 16.1.2 illustrates the idea. In equations,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{16.1.2}$$

As indicated in the error term, this symmetrization cancels out the first-order error term, making the method *second order*. [A method is conventionally called nth order if its error term is $O(h^{n+1})$.] In fact, (16.1.2) is called the *second-order Runge-Kutta* or *midpoint* method.

We needn't stop there. There are many ways to evaluate the right-hand side $f(x, y)$ that all agree to first order, but that have different coefficients of higher-order error terms. Adding up the right combination of these, we can eliminate the error terms order by order. That is the basic idea of the Runge-Kutta method. Abramowitz and Stegun [1], and Gear [2], give various specific formulas that derive from this basic

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.

Lambert, J. 1973, *Computational Methods in Ordinary Differential Equations* (New York: Wiley).

Lapidus, L., and Seinfeld, J. 1971, *Numerical Solution of Ordinary Differential Equations* (New York: Academic Press).

16.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{16.1.1}$$

which advances a solution from x_n to $x_{n+1} \equiv x_n + h$. The formula is unsymmetrical: It advances the solution through an interval h, but uses derivative information only at the beginning of that interval (see Figure 16.1.1). That means (and you can verify by expansion in power series) that the step's error is only one power of h smaller than the correction, i.e $O(h^2)$ added to (16.1.1).

There are several reasons that Euler's method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other, fancier, methods run at the equivalent stepsize, and (ii) neither is it very stable (see §16.6 below).

Consider, however, the use of a step like (16.1.1) to take a "trial" step to the midpoint of the interval. Then use the value of both x and y at that midpoint to compute the "real" step across the whole interval. Figure 16.1.2 illustrates the idea. In equations,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{16.1.2}$$

As indicated in the error term, this symmetrization cancels out the first-order error term, making the method *second order*. [A method is conventionally called nth order if its error term is $O(h^{n+1})$.] In fact, (16.1.2) is called the *second-order Runge-Kutta* or *midpoint* method.

We needn't stop there. There are many ways to evaluate the right-hand side $f(x, y)$ that all agree to first order, but that have different coefficients of higher-order error terms. Adding up the right combination of these, we can eliminate the error terms order by order. That is the basic idea of the Runge-Kutta method. Abramowitz and Stegun [1], and Gear [2], give various specific formulas that derive from this basic

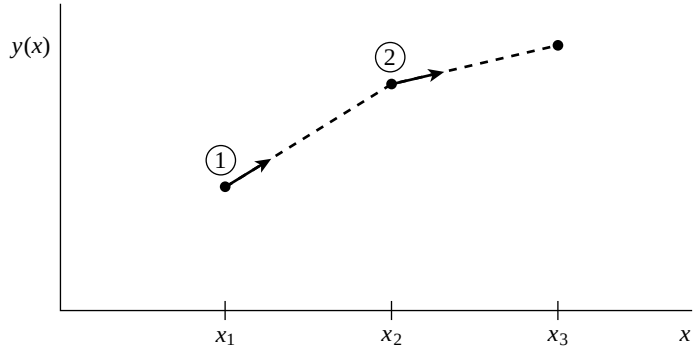


Figure 16.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

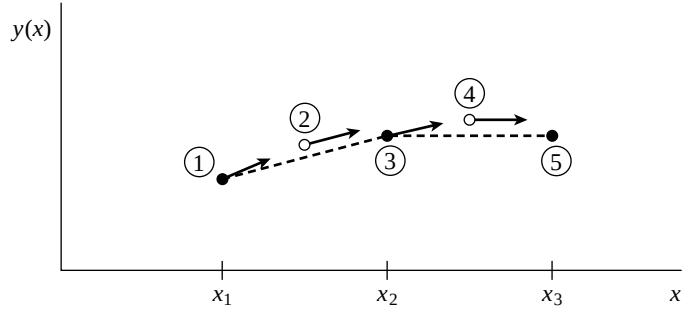


Figure 16.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

idea. By far the most often used is the classical *fourth-order Runge-Kutta formula*, which has a certain sleekness of organization about it:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \end{aligned} \tag{16.1.3}$$

The fourth-order Runge-Kutta method requires four evaluations of the right-hand side per step h (see Figure 16.1.3). This will be superior to the midpoint method (16.1.2) *if* at least twice as large a step is possible with (16.1.3) for the same accuracy. Is that so? The answer is: often, perhaps even usually, but surely not always! This takes us back to a central theme, namely that *high order* does not always mean *high accuracy*. The statement "fourth-order Runge-Kutta is generally superior to second-order" is a true one, but you should recognize it as a statement about the

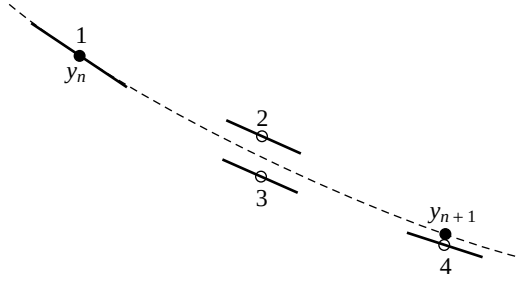


Figure 16.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

contemporary practice of science rather than as a statement about strict mathematics. That is, it reflects the nature of the problems that contemporary scientists like to solve.

For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. Keep in mind, however, that the old workhorse's last trip may well be to take you to the poorhouse: Bulirsch-Stoer or predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields. However, even the old workhorse is more nimble with new horseshoes. In §16.2 we will give a modern implementation of a Runge-Kutta method that is quite competitive as long as very high accuracy is not required. An excellent discussion of the pitfalls in constructing a good Runge-Kutta code is given in [3].

Here is the routine for carrying out one classical Runge-Kutta step on a set of `n` differential equations. You input the values of the independent variables, and you get out new values which are stepped by a stepsize `h` (which can be positive or negative). You will notice that the routine requires you to supply not only function `derivs` for calculating the right-hand side, but also values of the derivatives at the starting point. Why not let the routine call `derivs` for this first value? The answer will become clear only in the next section, but in brief is this: This call may not be your only one with these starting conditions. You may have taken a previous step with too large a stepsize, and this is your replacement. In that case, you do not want to call `derivs` unnecessarily at the start. Note that the routine that follows has, therefore, only three calls to `derivs`.

```
SUBROUTINE rk4(y,dydx,n,x,h,yout,derivs)
INTEGER n,NMAX
REAL h,x,dydx(n),y(n),yout(n)
EXTERNAL derivs
PARAMETER (NMAX=50)             Set to the maximum number of functions.
    Given values for the variables y(1:n) and their derivatives dydx(1:n) known at x, use
    the fourth-order Runge-Kutta method to advance the solution over an interval h and return
    the incremented variables as yout(1:n), which need not be a distinct array from y. The
    user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.
INTEGER i
REAL h6,hh,xh,dym(NMAX),dyt(NMAX),yt(NMAX)
hh=h*0.5
h6=h/6.
xh=x+hh
```

```
      do 11 i=1,n                    First step.
        yt(i)=y(i)+hh*dydx(i)
      enddo 11
      call derivs(xh,yt,dyt)         Second step.
      do 12 i=1,n
        yt(i)=y(i)+hh*dyt(i)
      enddo 12
      call derivs(xh,yt,dym)         Third step.
      do 13 i=1,n
        yt(i)=y(i)+h*dym(i)
        dym(i)=dyt(i)+dym(i)
      enddo 13
      call derivs(x+h,yt,dyt)        Fourth step.
      do 14 i=1,n                    Accumulate increments with proper weights.
        yout(i)=y(i)+h6*(dydx(i)+dyt(i)+2.*dym(i))
      enddo 14
      return
      END
```

The Runge-Kutta method treats every step in a sequence of steps in identical manner. Prior behavior of a solution is not used in its propagation. This is mathematically proper, since any point along the trajectory of an ordinary differential equation can serve as an initial point. The fact that all steps are treated identically also makes it easy to incorporate Runge-Kutta into relatively simple "driver" schemes.

We consider adaptive stepsize control, discussed in the next section, an essential for serious computing. Occasionally, however, you just want to tabulate a function at equally spaced intervals, and without particularly high accuracy. In the most common case, you want to produce a graph of the function. Then all you need may be a simple driver program that goes from an initial x_s to a final x_f in a specified number of steps. To check accuracy, double the number of steps, repeat the integration, and compare results. This approach surely does not minimize computer time, and it can fail for problems whose nature *requires* a variable stepsize, but it may well minimize user effort. On small problems, this may be the paramount consideration.

Here is such a driver, self-explanatory, which tabulates the integrated functions in a common block `path`.

```
      SUBROUTINE rkdumb(vstart,nvar,x1,x2,nstep,derivs)
      INTEGER nstep,nvar,NMAX,NSTPMX
      PARAMETER (NMAX=50,NSTPMX=200)                  Maximum number of functions and
      REAL x1,x2,vstart(nvar),xx(NSTPMX),y(NMAX,NSTPMX)   maximum number of values to
      EXTERNAL derivs                                     be stored.
      COMMON /path/ xx,y             Storage of results.
C     USES rk4
        Starting from initial values vstart(1:nvar) known at x1 use fourth-order Runge-Kutta to
        advance nstep equal increments to x2. The user-supplied subroutine derivs(x,v,dvdx)
        evaluates derivatives. Results are stored in the common block path. Be sure to dimension
        the common block appropriately.
      INTEGER i,k
      REAL h,x,dv(NMAX),v(NMAX)
      do 11 i=1,nvar                 Load starting values.
        v(i)=vstart(i)
        y(i,1)=v(i)
      enddo 11
      xx(1)=x1
      x=x1
      h=(x2-x1)/nstep
      do 13 k=1,nstep                Take nstep steps.
        call derivs(x,v,dv)
```

```
        call rk4(v,dv,nvar,x,h,v,derivs)
        if(x+h.eq.x)pause 'stepsize not significant in rkdumb'
        x=x+h
        xx(k+1)=x                    Store intermediate steps.
        do 12 i=1,nvar
            y(i,k+1)=v(i)
        enddo 12
    enddo 13
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.5. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121. [3]

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill), §9.2.

16.2 Adaptive Stepsize Control for Runge-Kutta

A good ODE integrator should exert some adaptive control over its own progress, making frequent changes in its stepsize. Usually the purpose of this adaptive stepsize control is to achieve some predetermined accuracy in the solution with minimum computational effort. Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through smooth uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more. Sometimes accuracy may be demanded not directly in the solution itself, but in some related conserved quantity that can be monitored.

Implementation of adaptive stepsize control requires that the stepping algorithm return information about its performance, most important, an estimate of its truncation error. In this section we will learn how such information can be obtained. Obviously, the calculation of this information will add to the computational overhead, but the investment will generally be repaid handsomely.

With fourth-order Runge-Kutta, the most straightforward technique by far is *step doubling* (see, e.g., [1]). We take each step twice, once as a full step, then, independently, as two half steps (see Figure 16.2.1). How much overhead is this, say in terms of the number of evaluations of the right-hand sides? Each of the three separate Runge-Kutta steps in the procedure requires 4 evaluations, but the single and double sequences share a starting point, so the total is 11. This is to be compared not to 4, but to 8 (the two half-steps), since — stepsize control aside — we are achieving the accuracy of the smaller (half) stepsize. The overhead cost is therefore a factor 1.375. What does it buy us?

```
        call rk4(v,dv,nvar,x,h,v,derivs)
        if(x+h.eq.x)pause 'stepsize not significant in rkdumb'
        x=x+h
        xx(k+1)=x                    Store intermediate steps.
        do 12 i=1,nvar
            y(i,k+1)=v(i)
        enddo 12
    enddo 13
    return
    END
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.5. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121. [3]

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill), §9.2.

16.2 Adaptive Stepsize Control for Runge-Kutta

A good ODE integrator should exert some adaptive control over its own progress, making frequent changes in its stepsize. Usually the purpose of this adaptive stepsize control is to achieve some predetermined accuracy in the solution with minimum computational effort. Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through smooth uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more. Sometimes accuracy may be demanded not directly in the solution itself, but in some related conserved quantity that can be monitored.

Implementation of adaptive stepsize control requires that the stepping algorithm return information about its performance, most important, an estimate of its truncation error. In this section we will learn how such information can be obtained. Obviously, the calculation of this information will add to the computational overhead, but the investment will generally be repaid handsomely.

With fourth-order Runge-Kutta, the most straightforward technique by far is *step doubling* (see, e.g., [1]). We take each step twice, once as a full step, then, independently, as two half steps (see Figure 16.2.1). How much overhead is this, say in terms of the number of evaluations of the right-hand sides? Each of the three separate Runge-Kutta steps in the procedure requires 4 evaluations, but the single and double sequences share a starting point, so the total is 11. This is to be compared not to 4, but to 8 (the two half-steps), since — stepsize control aside — we are achieving the accuracy of the smaller (half) stepsize. The overhead cost is therefore a factor 1.375. What does it buy us?

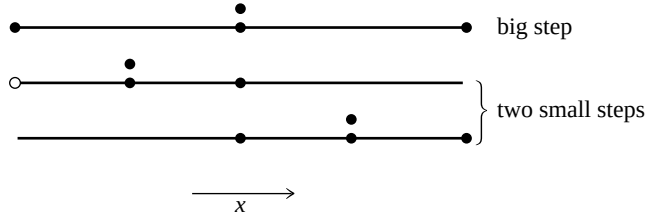


Figure 16.2.1. Step-doubling as a means for adaptive stepsize control in fourth-order Runge-Kutta. Points where the derivative is evaluated are shown as filled circles. The open circle represents the same derivatives as the filled circle immediately above it, so the total number of evaluations is 11 per two steps. Comparing the accuracy of the big step with the two small steps gives a criterion for adjusting the stepsize on the next step, or for rejecting the current step as inaccurate.

Let us denote the exact solution for an advance from x to $x + 2h$ by $y(x + 2h)$ and the two approximate solutions by y_1 (one step $2h$) and y_2 (2 steps each of size h). Since the basic method is fourth order, the true solution and the two numerical approximations are related by

$$\begin{aligned} y(x + 2h) &= y_1 + (2h)^5\phi + O(h^6) + \dots \\ y(x + 2h) &= y_2 + 2(h^5)\phi + O(h^6) + \dots \end{aligned} \tag{16.2.1}$$

where, to order h^5, the value ϕ remains constant over the step. [Taylor series expansion tells us the ϕ is a number whose order of magnitude is $y^{(5)}(x)/5!$.] The first expression in (16.2.1) involves $(2h)^5$ since the stepsize is $2h$, while the second expression involves $2(h^5)$ since the error on each step is $h^5\phi$. The difference between the two numerical estimates is a convenient indicator of truncation error

$$\Delta \equiv y_2 - y_1 \tag{16.2.2}$$

It is this difference that we shall endeavor to keep to a desired degree of accuracy, neither too large nor too small. We do this by adjusting h.

It might also occur to you that, ignoring terms of order h^6 and higher, we can solve the two equations in (16.2.1) to improve our numerical estimate of the true solution $y(x + 2h)$, namely,

$$y(x + 2h) = y_2 + \frac{\Delta}{15} + O(h^6) \tag{16.2.3}$$

This estimate is accurate to *fifth order*, one order higher than the original Runge-Kutta steps. However, we can't have our cake and eat it: (16.2.3) may be fifth-order accurate, but we have no way of monitoring *its* truncation error. Higher order is not always higher accuracy! Use of (16.2.3) rarely does harm, but we have no way of directly knowing whether it is doing any good. Therefore we should use Δ as the error estimate and take as "gravy" any additional accuracy gain derived from (16.2.3). In the technical literature, use of a procedure like (16.2.3) is called "local extrapolation."

An alternative stepsize adjustment algorithm is based on the *embedded Runge-Kutta formulas*, originally invented by Fehlberg. An interesting fact about Runge-Kutta formulas is that for orders M higher than four, more than M function

evaluations (though never more than $M + 2$) are required. This accounts for the popularity of the classical fourth-order method: It seems to give the most bang for the buck. However, Fehlberg discovered a fifth-order method with six function evaluations where another combination of the six functions gives a fourth-order method. The difference between the two estimates of $y(x + h)$ can then be used as an estimate of the truncation error to adjust the stepsize. Since Fehlberg's original formula, several other embedded Runge-Kutta formulas have been found.

Many practitioners were at one time wary of the robustness of Runge-Kutta-Fehlberg methods. The feeling was that using the same evaluation points to advance the function and to estimate the error was riskier than step-doubling, where the error estimate is based on independent function evaluations. However, experience has shown that this concern is not a problem in practice. Accordingly, embedded Runge-Kutta formulas, which are roughly a factor of two more efficient, have superseded algorithms based on step-doubling.

The general form of a fifth-order Runge-Kutta formula is

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + a_2 h, y_n + b_{21} k_1) \\ &\cdots \\ k_6 &= hf(x_n + a_6 h, y_n + b_{61} k_1 + \cdots + b_{65} k_5) \\ y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O(h^6) \end{aligned} \tag{16.2.4}$$

The embedded fourth-order formula is

$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 + O(h^5) \tag{16.2.5}$$

and so the error estimate is

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (c_i - c^*_i) k_i \tag{16.2.6}$$

The particular values of the various constants that we favor are those found by Cash and Karp [2], and given in the accompanying table. These give a more efficient method than Fehlberg's original values, with somewhat better error properties.

Now that we know, at least approximately, what our error is, we need to consider how to keep it within desired bounds. What is the relation between Δ and h? According to (16.2.4) – (16.2.5), Δ scales as h^5. If we take a step h_1 and produce an error Δ_1, therefore, the step h_0 that *would have given* some other value Δ_0 is readily estimated as

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \tag{16.2.7}$$

Henceforth we will let Δ_0 denote the *desired* accuracy. Then equation (16.2.7) is used in two ways: If Δ_1 is larger than Δ_0 in magnitude, the equation tells how much to decrease the stepsize *when we retry the present (failed) step.* If Δ_1 is

Cash-Karp Parameters for Embedded Runga-Kutta Method

i	a_i	b_{ij}					c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j =$		1	2	3	4	5		

smaller than Δ_0, on the other hand, then the equation tells how much we can safely increase the stepsize *for the next step*. Local extrapolation consists in accepting the fifth order value y_{n+1}, even though the error estimate actually applies to the fourth order value y^*_{n+1}.

Our notation hides the fact that Δ_0 is actually a vector of desired accuracies, one for each equation in the set of ODEs. In general, our accuracy requirement will be that all equations are within their respective allowed errors. In other words, we will rescale the stepsize according to the needs of the "worst-offender" equation.

How is Δ_0, the desired accuracy, related to some looser prescription like "get a solution good to one part in 10^6"? That can be a subtle question, and it depends on exactly what your application is! You may be dealing with a set of equations whose dependent variables differ enormously in magnitude. In that case, you probably want to use fractional errors, $\Delta_0 = \epsilon y$, where ϵ is the number like 10^{-6} or whatever. On the other hand, you may have oscillatory functions that pass through zero but are bounded by some maximum values. In that case you probably want to set Δ_0 equal to ϵ times those maximum values.

A convenient way to fold these considerations into a generally useful stepper routine is this: One of the arguments of the routine will of course be the vector of dependent variables at the beginning of a proposed step. Call that `y(1:n)`. Let us require the user to specify for each step another, corresponding, vector argument `yscal(1:n)`, and also an overall tolerance level `eps`. Then the desired accuracy for the ith equation will be taken to be

$$\Delta_0 = \texttt{eps} \times \texttt{yscal(i)} \tag{16.2.8}$$

If you desire constant fractional errors, plug `y` into the `yscal` calling slot (no need to copy the values into a different array). If you desire constant absolute errors relative to some maximum values, set the elements of `yscal` equal to those maximum values. A useful "trick" for getting constant fractional errors *except* "very" near zero crossings is to set `yscal(i)` equal to $|\texttt{y(i)}| + |h \times \texttt{dydx(i)}|$. (The routine `odeint`, below, does this.)

Here is a more technical point. We have to consider one additional possibility for `yscal`. The error criteria mentioned thus far are "local," in that they bound the error of each step individually. In some applications you may be unusually sensitive

about a "global" accumulation of errors, from beginning to end of the integration and in the worst possible case where the errors all are presumed to add with the same sign. Then, the smaller the stepsize h, the smaller the value Δ_0 that you will need to impose. Why? Because there will be *more steps* between your starting and ending values of x. In such cases you will want to set `yscal` proportional to h, typically to something like

$$\Delta_0 = \epsilon h \times \texttt{dydx(i)} \tag{16.2.9}$$

This enforces fractional accuracy ϵ not on the values of y but (much more stringently) on the *increments* to those values at each step. But now look back at (16.2.7). If Δ_0 has an implicit scaling with h, then the exponent 0.20 is no longer correct: When the stepsize is reduced from a too-large value, the new predicted value h_1 will fail to meet the desired accuracy when `yscal` is also altered to this new h_1 value. Instead of $0.20 = 1/5$, we must scale by the exponent $0.25 = 1/4$ for things to work out.

The exponents 0.20 and 0.25 are not really very different. This motivates us to adopt the following pragmatic approach, one that frees us from having to know in advance whether or not you, the user, plan to scale your `yscal`'s with stepsize. Whenever we decrease a stepsize, let us use the larger value of the exponent (whether we need it or not!), and whenever we increase a stepsize, let us use the smaller exponent. Furthermore, because our estimates of error are not exact, but only accurate to the leading order in h, we are advised to put in a safety factor S which is a few percent smaller than unity. Equation (16.2.7) is thus replaced by

$$h_0 = \begin{cases} Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.20} & \Delta_0 \geq \Delta_1 \\ Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{16.2.10}$$

We have found this prescription to be a reliable one in practice.

Here, then, is a stepper program that takes one "quality-controlled" Runge-Kutta step.

```
      SUBROUTINE rkqs(y,dydx,n,x,htry,eps,yscal,hdid,hnext,derivs)
      INTEGER n,NMAX
      REAL eps,hdid,hnext,htry,x,dydx(n),y(n),yscal(n)
      EXTERNAL derivs
      PARAMETER (NMAX=50)                     Maximum number of equations.
C     USES derivs,rkck
        Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy
        and adjust stepsize. Input are the dependent variable vector y(1:n) and its derivative
        dydx(1:n) at the starting value of the independent variable x. Also input are the stepsize
        to be attempted htry, the required accuracy eps, and the vector yscal(1:n) against
        which the error is scaled. On output, y and x are replaced by their new values, hdid is the
        stepsize that was actually accomplished, and hnext is the estimated next stepsize. derivs
        is the user-supplied subroutine that computes the right-hand side derivatives.
      INTEGER i
      REAL errmax,h,htemp,xnew,yerr(NMAX),ytemp(NMAX),SAFETY,PGROW,
     *     PSHRNK,ERRCON
      PARAMETER (SAFETY=0.9,PGROW=-.2,PSHRNK=-.25,ERRCON=1.89e-4)
        The value ERRCON equals (5/SAFETY)**(1/PGROW), see use below.
      h=htry                                  Set stepsize to the initial trial value.
1     call rkck(y,dydx,n,x,h,ytemp,yerr,derivs)          Take a step.
```

```
      errmax=0.                                  Evaluate accuracy.
      do 11 i=1,n
        errmax=max(errmax,abs(yerr(i)/yscal(i)))
      enddo 11
      errmax=errmax/eps                          Scale relative to required tolerance.
      if(errmax.gt.1.)then                       Truncation error too large, reduce stepsize.
        htemp=SAFETY*h*(errmax**PSHRNK)
        h=sign(max(abs(htemp),0.1*abs(h)),h)       No more than a factor of 10.
        xnew=x+h
        if(xnew.eq.x)pause 'stepsize underflow in rkqs'
        goto 1                                   For another try.
      else                                       Step succeeded. Compute size of next step.
        if(errmax.gt.ERRCON)then
          hnext=SAFETY*h*(errmax**PGROW)
        else                                     No more than a factor of 5 increase.
          hnext=5.*h
        endif
        hdid=h
        x=x+h
        do 12 i=1,n
          y(i)=ytemp(i)
        enddo 12
        return
      endif
      END
```

The routine `rkqs` calls the routine `rkck` to take a Cash-Karp Runge-Kutta step:

```
      SUBROUTINE rkck(y,dydx,n,x,h,yout,yerr,derivs)
      INTEGER n,NMAX
      REAL h,x,dydx(n),y(n),yerr(n),yout(n)
      EXTERNAL derivs
      PARAMETER (NMAX=50)                        Set to the maximum number of functions.
C     USES derivs
```

Given values for `n` variables `y` and their derivatives `dydx` known at `x`, use the fifth-order Cash-Karp Runge-Kutta method to advance the solution over an interval `h` and return the incremented variables as `yout`. Also return an estimate of the local truncation error in `yout` using the embedded fourth-order method. The user supplies the subroutine `derivs(x,y,dydx)`, which returns derivatives `dydx` at `x`.

```
      INTEGER i
      REAL ak2(NMAX),ak3(NMAX),ak4(NMAX),ak5(NMAX),ak6(NMAX),
     *     ytemp(NMAX),A2,A3,A4,A5,A6,B21,B31,B32,B41,B42,B43,B51,
     *     B52,B53,B54,B61,B62,B63,B64,B65,C1,C3,C4,C6,DC1,DC3,
     *     DC4,DC5,DC6
      PARAMETER (A2=.2,A3=.3,A4=.6,A5=1.,A6=.875,B21=.2,B31=3./40.,
     *     B32=9./40.,B41=.3,B42=-.9,B43=1.2,B51=-11./54.,B52=2.5,
     *     B53=-70./27.,B54=35./27.,B61=1631./55296.,B62=175./512.,
     *     B63=575./13824.,B64=44275./110592.,B65=253./4096.,
     *     C1=37./378.,C3=250./621.,C4=125./594.,C6=512./1771.,
     *     DC1=C1-2825./27648.,DC3=C3-18575./48384.,
     *     DC4=C4-13525./55296.,DC5=-277./14336.,DC6=C6-.25)
      do 11 i=1,n                                First step.
        ytemp(i)=y(i)+B21*h*dydx(i)
      enddo 11
      call derivs(x+A2*h,ytemp,ak2)              Second step.
      do 12 i=1,n
        ytemp(i)=y(i)+h*(B31*dydx(i)+B32*ak2(i))
      enddo 12
      call derivs(x+A3*h,ytemp,ak3)              Third step.
      do 13 i=1,n
        ytemp(i)=y(i)+h*(B41*dydx(i)+B42*ak2(i)+B43*ak3(i))
```

```
      enddo 13
      call derivs(x+A4*h,ytemp,ak4)       Fourth step.
      do 14 i=1,n
        ytemp(i)=y(i)+h*(B51*dydx(i)+B52*ak2(i)+B53*ak3(i)+
     *       B54*ak4(i))
      enddo 14
      call derivs(x+A5*h,ytemp,ak5)       Fifth step.
      do 15 i=1,n
        ytemp(i)=y(i)+h*(B61*dydx(i)+B62*ak2(i)+B63*ak3(i)+
     *       B64*ak4(i)+B65*ak5(i))
      enddo 15
      call derivs(x+A6*h,ytemp,ak6)       Sixth step.
      do 16 i=1,n                         Accumulate increments with proper weights.
        yout(i)=y(i)+h*(C1*dydx(i)+C3*ak3(i)+C4*ak4(i)+
     *       C6*ak6(i))
      enddo 16
      do 17 i=1,n
        Estimate error as difference between fourth and fifth order methods.
        yerr(i)=h*(DC1*dydx(i)+DC3*ak3(i)+DC4*ak4(i)+DC5*ak5(i)
     *       +DC6*ak6(i))
      enddo 17
      return
      END
```

Noting that the above routines are all in single precision, don't be too greedy in specifying eps. The punishment for excessive greediness is interesting and worthy of Gilbert and Sullivan's *Mikado*: The routine can always achieve an apparent *zero* error by making the stepsize so small that quantities of order hy' add to quantities of order y as if they were zero. Then the routine chugs happily along taking infinitely many infinitesimal steps and never changing the dependent variables one iota. (You guard against this catastrophic loss of your computer budget by signaling on abnormally small stepsizes or on the dependent variable vector remaining unchanged from step to step. On a personal workstation you guard against it by not taking too long a lunch hour while your program is running.)

Here is a full-fledged "driver" for Runge-Kutta with adaptive stepsize control. We warmly recommend this routine, or one like it, for a variety of problems, notably including garden-variety ODEs or sets of ODEs, and definite integrals (augmenting the methods of Chapter 4). For storage of intermediate results (if you desire to inspect them) we assume a common block path, which can hold up to KMAXX steps. Because steps occur at unequal intervals results are stored only at intervals greater than dxsav. Also in the block is kmax, indicating the number of steps that can be stored. If kmax=0 there is no intermediate storage, and the rest of the common block need not exist. Otherwise you should set kmax = KMAXX. Storage of steps stops if kmax is exceeded, except that the ending values are always stored. Again, these controls are merely indicative of what you might need. The routine odeint should be customized to the problem at hand.

```
      SUBROUTINE odeint(ystart,nvar,x1,x2,eps,h1,hmin,nok,nbad,derivs,rkqs)
      INTEGER nbad,nok,nvar,KMAXX,MAXSTP,NMAX
      REAL eps,h1,hmin,x1,x2,ystart(nvar),TINY
      EXTERNAL derivs,rkqs
      PARAMETER (MAXSTP=10000,NMAX=50,KMAXX=200,TINY=1.e-30)
        Runge-Kutta driver with adaptive stepsize control. Integrate the starting values ystart(1:nvar)
        from x1 to x2 with accuracy eps, storing intermediate results in the common block /path/.
        h1 should be set as a guessed first stepsize, hmin as the minimum allowed stepsize (can
        be zero). On output nok and nbad are the number of good and bad (but retried and
```

fixed) steps taken, and ystart is replaced by values at the end of the integration interval. derivs is the user-supplied subroutine for calculating the right-hand side derivative, while rkqs is the name of the stepper routine to be used. /path/ contains its own information about how often an intermediate value is to be stored.

```
      INTEGER i,kmax,kount,nstp
      REAL dxsav,h,hdid,hnext,x,xsav,dydx(NMAX),xp(KMAXX),y(NMAX),
     *     yp(NMAX,KMAXX),yscal(NMAX)
      COMMON /path/ kmax,kount,dxsav,xp,yp
```

User storage for intermediate results. Preset dxsav and kmax.

```
      x=x1
      h=sign(h1,x2-x1)
      nok=0
      nbad=0
      kount=0
      do 11 i=1,nvar
        y(i)=ystart(i)
      enddo 11
      if (kmax.gt.0) xsav=x-2.*dxsav                    Assures storage of first step.
      do 16 nstp=1,MAXSTP                               Take at most MAXSTP steps.
        call derivs(x,y,dydx)
        do 12 i=1,nvar
```

Scaling used to monitor accuracy. This general-purpose choice can be modified if need be.

```
          yscal(i)=abs(y(i))+abs(h*dydx(i))+TINY
        enddo 12
        if(kmax.gt.0)then
          if(abs(x-xsav).gt.abs(dxsav)) then            Store intermediate results.
            if(kount.lt.kmax-1)then
              kount=kount+1
              xp(kount)=x
              do 13 i=1,nvar
                yp(i,kount)=y(i)
              enddo 13
              xsav=x
            endif
          endif
        endif
        if((x+h-x2)*(x+h-x1).gt.0.) h=x2-x              If stepsize can overshoot, decrease.
        call rkqs(y,dydx,nvar,x,h,eps,yscal,hdid,hnext,derivs)
        if(hdid.eq.h)then
          nok=nok+1
        else
          nbad=nbad+1
        endif
        if((x-x2)*(x2-x1).ge.0.)then                    Are we done?
          do 14 i=1,nvar
            ystart(i)=y(i)
          enddo 14
          if(kmax.ne.0)then
            kount=kount+1                               Save final step.
            xp(kount)=x
            do 15 i=1,nvar
              yp(i,kount)=y(i)
            enddo 15
          endif
          return                                        Normal exit.
        endif
        if(abs(hnext).lt.hmin) pause 'stepsize smaller than minimum in odeint'
        h=hnext
      enddo 16
      pause 'too many steps in odeint'
      return
      END
```

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Cash, J.R., and Karp, A.H. 1990, *ACM Transactions on Mathematical Software*, vol. 16, pp. 201–222. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall).

16.3 Modified Midpoint Method

This section discusses the *modified midpoint method*, which advances a vector of dependent variables $y(x)$ from a point x to a point $x + H$ by a sequence of n substeps each of size h,

$$h = H/n \tag{16.3.1}$$

In principle, one could use the modified midpoint method in its own right as an ODE integrator. In practice, the method finds its most important application as a part of the more powerful Bulirsch-Stoer technique, treated in §16.4. You can therefore consider this section as a preamble to §16.4.

The number of right-hand side evaluations required by the modified midpoint method is $n + 1$. The formulas for the method are

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x + mh, z_m) \qquad \text{for} \quad m = 1, 2, \ldots, n-1 \\ y(x + H) \approx y_n &\equiv \frac{1}{2}[z_n + z_{n-1} + hf(x + H, z_n)] \end{aligned} \tag{16.3.2}$$

Here the z's are intermediate approximations which march along in steps of h, while y_n is the final approximation to $y(x + H)$. The method is basically a "centered difference" or "midpoint" method (compare equation 16.1.2), except at the first and last points. Those give the qualifier "modified."

The modified midpoint method is a second-order method, like (16.1.2), but with the advantage of requiring (asymptotically for large n) only one derivative evaluation per step h instead of the two required by second-order Runge-Kutta. Perhaps there are applications where the simplicity of (16.3.2), easily coded in-line in some other program, recommends it. In general, however, use of the modified midpoint method by itself will be dominated by the embedded Runge-Kutta method with adaptive stepsize control, as implemented in the preceding section.

The usefulness of the modified midpoint method to the Bulirsch-Stoer technique (§16.4) derives from a "deep" result about equations (16.3.2), due to Gragg. It turns

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Cash, J.R., and Karp, A.H. 1990, *ACM Transactions on Mathematical Software*, vol. 16, pp. 201–222. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall).

16.3 Modified Midpoint Method

This section discusses the *modified midpoint method*, which advances a vector of dependent variables $y(x)$ from a point x to a point $x + H$ by a sequence of n substeps each of size h,

$$h = H/n \tag{16.3.1}$$

In principle, one could use the modified midpoint method in its own right as an ODE integrator. In practice, the method finds its most important application as a part of the more powerful Bulirsch-Stoer technique, treated in §16.4. You can therefore consider this section as a preamble to §16.4.

The number of right-hand side evaluations required by the modified midpoint method is $n + 1$. The formulas for the method are

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x + mh, z_m) \qquad \text{for} \quad m = 1, 2, \ldots, n-1 \\ y(x + H) \approx y_n &\equiv \frac{1}{2}[z_n + z_{n-1} + hf(x + H, z_n)] \end{aligned} \tag{16.3.2}$$

Here the z's are intermediate approximations which march along in steps of h, while y_n is the final approximation to $y(x + H)$. The method is basically a "centered difference" or "midpoint" method (compare equation 16.1.2), except at the first and last points. Those give the qualifier "modified."

The modified midpoint method is a second-order method, like (16.1.2), but with the advantage of requiring (asymptotically for large n) only one derivative evaluation per step h instead of the two required by second-order Runge-Kutta. Perhaps there are applications where the simplicity of (16.3.2), easily coded in-line in some other program, recommends it. In general, however, use of the modified midpoint method by itself will be dominated by the embedded Runge-Kutta method with adaptive stepsize control, as implemented in the preceding section.

The usefulness of the modified midpoint method to the Bulirsch-Stoer technique (§16.4) derives from a "deep" result about equations (16.3.2), due to Gragg. It turns

out that the error of (16.3.2), expressed as a power series in h, the stepsize, contains only *even* powers of h,

$$y_n - y(x+H) = \sum_{i=1}^{\infty} \alpha_i h^{2i} \tag{16.3.3}$$

where H is held constant, but h changes by varying n in (16.3.1). The importance of this even power series is that, if we play our usual tricks of combining steps to knock out higher-order error terms, we can gain *two* orders at a time!

For example, suppose n is even, and let $y_{n/2}$ denote the result of applying (16.3.1) and (16.3.2) with half as many steps, $n \to n/2$. Then the estimate

$$y(x+H) \approx \frac{4y_n - y_{n/2}}{3} \tag{16.3.4}$$

is *fourth-order* accurate, the same as fourth-order Runge-Kutta, but requires only about 1.5 derivative evaluations per step h instead of Runge-Kutta's 4 evaluations. Don't be too anxious to implement (16.3.4), since we will soon do even better.

Now would be a good time to look back at the routine `qsimp` in §4.2, and especially to compare equation (4.2.4) with equation (16.3.4) above. You will see that the transition in Chapter 4 to the idea of Richardson extrapolation, as embodied in Romberg integration of §4.3, is exactly analogous to the transition in going from this section to the next one.

Here is the routine that implements the modified midpoint method, which will be used below.

```
SUBROUTINE mmid(y,dydx,nvar,xs,htot,nstep,yout,derivs)
INTEGER nstep,nvar,NMAX
REAL htot,xs,dydx(nvar),y(nvar),yout(nvar)
EXTERNAL derivs
PARAMETER (NMAX=50)
    Modified midpoint step. Dependent variable vector y(1:nvar) and its derivative vector
    dydx(1:nvar) are input at xs. Also input is htot, the total step to be made, and nstep,
    the number of substeps to be used. The output is returned as yout(1:nvar), which need
    not be a distinct array from y; if it is distinct, however, then y and dydx are returned
    undamaged.
INTEGER i,n
REAL h,h2,swap,x,ym(NMAX),yn(NMAX)
h=htot/nstep                    Stepsize this trip.
do 11 i=1,nvar
    ym(i)=y(i)
    yn(i)=y(i)+h*dydx(i)        First step.
enddo 11
x=xs+h
call derivs(x,yn,yout)          Will use yout for temporary storage of derivatives.
h2=2.*h
do 13 n=2,nstep                 General step.
    do 12 i=1,nvar
        swap=ym(i)+h2*yout(i)
        ym(i)=yn(i)
        yn(i)=swap
    enddo 12
    x=x+h
    call derivs(x,yn,yout)
```

```
enddo 13
do 14 i=1,nvar                    Last step.
    yout(i)=0.5*(ym(i)+yn(i)+h*yout(i))
enddo 14
return
END
```

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §7.2.12.

16.4 Richardson Extrapolation and the Bulirsch-Stoer Method

The techniques described in this section are not for differential equations containing nonsmooth functions. For example, you might have a differential equation whose right-hand side involves a function that is evaluated by table look-up and interpolation. If so, go back to Runge-Kutta with adaptive stepsize choice: That method does an excellent job of feeling its way through rocky or discontinuous terrain. It is also an excellent choice for quick-and-dirty, low-accuracy solution of a set of equations. A second warning is that the techniques in this section are not particularly good for differential equations that have singular points *inside* the interval of integration. A regular solution must tiptoe very carefully across such points. Runge-Kutta with adaptive stepsize can sometimes effect this; more generally, there are special techniques available for such problems, beyond our scope here.

Apart from those two caveats, we believe that the Bulirsch-Stoer method, discussed in this section, is the best known way to obtain high-accuracy solutions to ordinary differential equations with minimal computational effort. (A possible exception, infrequently encountered in practice, is discussed in §16.7.)

Three key ideas are involved. The first is *Richardson's deferred approach to the limit*, which we already met in §4.3 on Romberg integration. The idea is to consider the final answer of a numerical calculation as itself being an analytic function (if a complicated one) of an adjustable parameter like the stepsize h. That analytic function can be probed by performing the calculation with various values of h, *none* of them being necessarily small enough to yield the accuracy that we desire. When we know enough about the function, we *fit* it to some analytic form, and then *evaluate* it at that mythical and golden point $h = 0$ (see Figure 16.4.1). Richardson extrapolation is a method for turning straw into gold! (Lead into gold for alchemist readers.)

The second idea has to do with what kind of fitting function is used. Bulirsch and Stoer first recognized the strength of *rational function extrapolation* in Richardson-type applications. That strength is to break the shackles of the power series and its limited radius of convergence, out only to the distance of the first pole in the complex

```
enddo 13
do 14 i=1,nvar                    Last step.
    yout(i)=0.5*(ym(i)+yn(i)+h*yout(i))
enddo 14
return
END
```

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §7.2.12.

16.4 Richardson Extrapolation and the Bulirsch-Stoer Method

The techniques described in this section are not for differential equations containing nonsmooth functions. For example, you might have a differential equation whose right-hand side involves a function that is evaluated by table look-up and interpolation. If so, go back to Runge-Kutta with adaptive stepsize choice: That method does an excellent job of feeling its way through rocky or discontinuous terrain. It is also an excellent choice for quick-and-dirty, low-accuracy solution of a set of equations. A second warning is that the techniques in this section are not particularly good for differential equations that have singular points *inside* the interval of integration. A regular solution must tiptoe very carefully across such points. Runge-Kutta with adaptive stepsize can sometimes effect this; more generally, there are special techniques available for such problems, beyond our scope here.

Apart from those two caveats, we believe that the Bulirsch-Stoer method, discussed in this section, is the best known way to obtain high-accuracy solutions to ordinary differential equations with minimal computational effort. (A possible exception, infrequently encountered in practice, is discussed in §16.7.)

Three key ideas are involved. The first is *Richardson's deferred approach to the limit*, which we already met in §4.3 on Romberg integration. The idea is to consider the final answer of a numerical calculation as itself being an analytic function (if a complicated one) of an adjustable parameter like the stepsize h. That analytic function can be probed by performing the calculation with various values of h, *none* of them being necessarily small enough to yield the accuracy that we desire. When we know enough about the function, we *fit* it to some analytic form, and then *evaluate* it at that mythical and golden point $h = 0$ (see Figure 16.4.1). Richardson extrapolation is a method for turning straw into gold! (Lead into gold for alchemist readers.)

The second idea has to do with what kind of fitting function is used. Bulirsch and Stoer first recognized the strength of *rational function extrapolation* in Richardson-type applications. That strength is to break the shackles of the power series and its limited radius of convergence, out only to the distance of the first pole in the complex

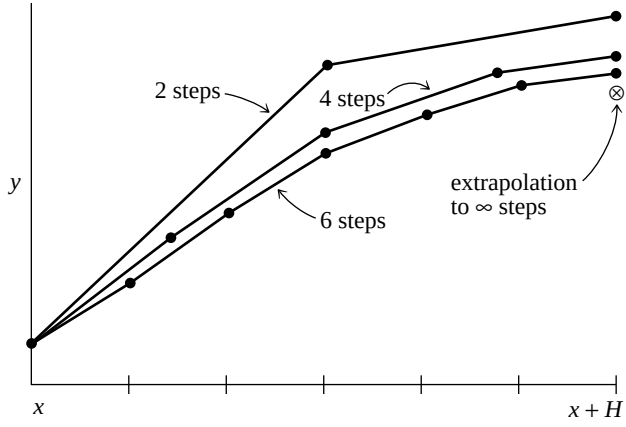


Figure 16.4.1. Richardson extrapolation as used in the Bulirsch-Stoer method. A large interval H is spanned by different sequences of finer and finer substeps. Their results are extrapolated to an answer that is supposed to correspond to infinitely fine substeps. In the Bulirsch-Stoer method, the integrations are done by the modified midpoint method, and the extrapolation technique is rational function or polynomial extrapolation.

plane. Rational function fits can remain good approximations to analytic functions even after the various terms in powers of h all have comparable magnitudes. In other words, h can be so large as to make the whole notion of the "order" of the method meaningless — and the method can still work superbly. Nevertheless, more recent experience suggests that for smooth problems straightforward polynomial extrapolation is slightly more efficient than rational function extrapolation. We will accordingly adopt polynomial extrapolation as the default, but the routine `bsstep` below allows easy substitution of one kind of extrapolation for the other. You might wish at this point to review §3.1–§3.2, where polynomial and rational function extrapolation were already discussed.

The third idea was discussed in the section before this one, namely to use a method whose error function is strictly even, allowing the rational function or polynomial approximation to be in terms of the variable h^2 instead of just h.

Put these ideas together and you have the *Bulirsch-Stoer method* [1]. A single Bulirsch-Stoer step takes us from x to $x + H$, where H is supposed to be quite a large — not at all infinitesimal — distance. That single step is a grand leap consisting of many (e.g., dozens to hundreds) substeps of modified midpoint method, which are then extrapolated to zero stepsize.

The sequence of separate attempts to cross the interval H is made with increasing values of n, the number of substeps. Bulirsch and Stoer originally proposed the sequence

$$n = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, \ldots, [n_j = 2n_{j-2}], \ldots \tag{16.4.1}$$

More recent work by Deuflhard [2,3] suggests that the sequence

$$n = 2, 4, 6, 8, 10, 12, 14, \ldots, [n_j = 2j], \ldots \tag{16.4.2}$$

is usually more efficient. For each step, we do not know in advance how far up this sequence we will go. After each successive n is tried, a polynomial extrapolation is

attempted. That extrapolation returns both extrapolated values and error estimates. If the errors are not satisfactory, we go higher in n. If they are satisfactory, we go on to the next step and begin anew with $n = 2$.

Of course there must be some upper limit, beyond which we conclude that there is some obstacle in our path in the interval H, so that we must reduce H rather than just subdivide it more finely. In the implementations below, the maximum number of n's to be tried is called KMAXX. For reasons described below we usually take this equal to 8; the 8th value of the sequence (16.4.2) is 16, so this is the maximum number of subdivisions of H that we allow.

We enforce error control, as in the Runge-Kutta method, by monitoring internal consistency, and adapting stepsize to match a prescribed bound on the local truncation error. Each new result from the sequence of modified midpoint integrations allows a tableau like that in §3.1 to be extended by one additional set of diagonals. The size of the new correction added at each stage is taken as the (conservative) error estimate. How should we use this error estimate to adjust the stepsize? The best strategy now known is due to Deuflhard [2,3]. For completeness we describe it here:

Suppose the absolute value of the error estimate returned from the kth column (and hence the $k+1$st row) of the extrapolation tableau is $\epsilon_{k+1,k}$. Error control is enforced by requiring

$$\epsilon_{k+1,k} < \epsilon \tag{16.4.3}$$

as the criterion for accepting the current step, where ϵ is the required tolerance. For the even sequence (16.4.2) the order of the method is $2k+1$:

$$\epsilon_{k+1,k} \sim H^{2k+1} \tag{16.4.4}$$

Thus a simple estimate of a new stepsize H_k to obtain convergence in a fixed column k would be

$$H_k = H \left(\frac{\epsilon}{\epsilon_{k+1,k}} \right)^{1/(2k+1)} \tag{16.4.5}$$

Which column k should we aim to achieve convergence in? Let's compare the work required for different k. Suppose A_k is the work to obtain row k of the extrapolation tableau, so A_{k+1} is the work to obtain column k. We will assume the work is dominated by the cost of evaluating the functions defining the right-hand sides of the differential equations. For n_k subdivisions in H, the number of function evaluations can be found from the recurrence

$$\begin{aligned} A_1 &= n_1 + 1 \\ A_{k+1} &= A_k + n_{k+1} \end{aligned} \tag{16.4.6}$$

The work per unit step to get column k is A_{k+1}/H_k, which we nondimensionalize with a factor of H and write as

$$W_k = \frac{A_{k+1}}{H_k} H \tag{16.4.7}$$

$$= A_{k+1} \left(\frac{\epsilon_{k+1,k}}{\epsilon} \right)^{1/(2k+1)} \tag{16.4.8}$$

The quantities W_k can be calculated during the integration. The optimal column index q is then defined by

$$W_q = \min_{k=1,\ldots,k_f} W_k \tag{16.4.9}$$

where k_f is the final column, in which the error criterion (16.4.3) was satisfied. The q determined from (16.4.9) defines the stepsize H_q to be used as the next basic stepsize, so that we can expect to get convergence in the optimal column q.

Two important refinements have to be made to the strategy outlined so far:

- If the current H is "too small," then k_f will be "too small," and so q remains "too small." It may be desirable to increase H and aim for convergence in a column $q > k_f$.
- If the current H is "too big," we may not converge at all on the current step and we will have to decrease H. We would like to detect this by monitoring the quantities $\epsilon_{k+1,k}$ for each k so we can stop the current step as soon as possible.

Deuflhard's prescription for dealing with these two problems uses ideas from communication theory to determine the "average expected convergence behavior" of the extrapolation. His model produces certain correction factors $\alpha(k,q)$ by which H_k is to be multiplied to try to get convergence in column q. The factors $\alpha(k,q)$ depend only on ϵ and the sequence $\{n_i\}$ and so can be computed once during initialization:

$$\alpha(k,q) = \epsilon^{\frac{A_{k+1}-A_{q+1}}{(2k+1)(A_{q+1}-A_1+1)}} \qquad \text{for} \quad k < q \tag{16.4.10}$$

with $\alpha(q,q) = 1$.

Now to handle the first problem, suppose convergence occurs in column $q = k_f$. Then rather than taking H_q for the next step, we might aim to increase the stepsize to get convergence in column $q+1$. Since we don't have H_{q+1} available from the computation, we estimate it as

$$H_{q+1} = H_q \alpha(q, q+1) \tag{16.4.11}$$

By equation (16.4.7) this replacement is efficient, i.e., reduces the work per unit step, if

$$\frac{A_{q+1}}{H_q} > \frac{A_{q+2}}{H_{q+1}} \tag{16.4.12}$$

or

$$A_{q+1}\alpha(q, q+1) > A_{q+2} \tag{16.4.13}$$

During initialization, this inequality can be checked for $q = 1, 2, \ldots$ to determine $k_{\max}$, the largest allowed column. Then when (16.4.12) is satisfied it will always be efficient to use H_{q+1}. (In practice we limit $k_{\max}$ to 8 even when ϵ is very small as there is very little further gain in efficiency whereas roundoff can become a problem.)

The problem of stepsize reduction is handled by computing stepsize estimates

$$\bar{H}_k \equiv H_k \alpha(k,q), \qquad k = 1, \ldots, q-1 \tag{16.4.14}$$

during the current step. The $\bar{H}$'s are estimates of the stepsize to get convergence in the optimal column q. If any $\bar{H}_k$ is "too small," we abandon the current step and restart using $\bar{H}_k$. The criterion of being "too small" is taken to be

$$H_k \alpha(k, q+1) < H \tag{16.4.15}$$

The α's satisfy $\alpha(k, q+1) > \alpha(k,q)$.

During the first step, when we have no information about the solution, the stepsize reduction check is made for all k. Afterwards, we test for convergence and for possible stepsize reduction only in an "order window"

$$\max(1, q-1) \le k \le \min(k_{\max}, q+1) \tag{16.4.16}$$

The rationale for the order window is that if convergence appears to occur for $k < q-1$ it is often spurious, resulting from some fortuitously small error estimate in the extrapolation. On the other hand, if you need to go beyond $k = q+1$ to obtain convergence, your local model of the convergence behavior is obviously not very good and you need to cut the stepsize and reestablish it.

In the routine `bsstep`, these various tests are actually carried out using quantities

$$\epsilon(k) \equiv \frac{H}{H_k} = \left(\frac{\epsilon_{k+1,k}}{\epsilon}\right)^{1/(2k+1)} \tag{16.4.17}$$

called `err(k)` in the code. As usual, we include a "safety factor" in the stepsize selection. This is implemented by replacing ϵ by 0.25ϵ. Other safety factors are explained in the program comments.

Note that while the optimal convergence column is restricted to increase by at most one on each step, a sudden drop in order is allowed by equation (16.4.9). This gives the method a degree of robustness for problems with discontinuities.

Let us remind you once again that *scaling* of the variables is often crucial for successful integration of differential equations. The scaling "trick" suggested in the discussion following equation (16.2.8) is a good general purpose choice, but not foolproof. Scaling by the maximum values of the variables is more robust, but requires you to have some prior information.

The following implementation of a Bulirsch-Stoer step has exactly the same calling sequence as the quality-controlled Runge-Kutta stepper `rkqs`. This means that the driver `odeint` in §16.2 can be used for Bulirsch-Stoer as well as Runge-Kutta: Just substitute `bsstep` for `rkqs` in `odeint`'s argument list. The routine `bsstep` calls `mmid` to take the modified midpoint sequences, and calls `pzextr`, given below, to do the polynomial extrapolation.

```
      SUBROUTINE bsstep(y,dydx,nv,x,htry,eps,yscal,hdid,hnext,derivs)
      INTEGER nv,NMAX,KMAXX,IMAX
      REAL eps,hdid,hnext,htry,x,dydx(nv),y(nv),yscal(nv),SAFE1,
     *     SAFE2,REDMAX,REDMIN,TINY,SCALMX
      PARAMETER (NMAX=50,KMAXX=8,IMAX=KMAXX+1,SAFE1=.25,SAFE2=.7,
     *     REDMAX=1.e-5,REDMIN=.7,TINY=1.e-30,SCALMX=.1)
C     USES derivs,mmid,pzextr
```

Bulirsch-Stoer step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector `y(1:nv)` and its derivative `dydx(1:nv)` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal(1:nv)` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is the user-supplied subroutine that computes the right-hand side derivatives. Be sure to set `htry` on successive steps to the value of `hnext` returned from the previous step, as is the case if the routine is called by `odeint`.

Parameters: `NMAX` is the maximum value of `nv`; `KMAXX` is the maximum row number used in the extrapolation; `IMAX` is the next row number; `SAFE1` and `SAFE2` are safety factors; `REDMAX` is the maximum factor used when a stepsize is reduced, `REDMIN` the minimum; `TINY` prevents division by zero; `1/SCALMX` is the maximum factor by which a stepsize can be increased.

```
      INTEGER i,iq,k,kk,km,kmax,kopt,nseq(IMAX)
      REAL eps1,epsold,errmax,fact,h,red,scale,work,wrkmin,xest,
     *     xnew,a(IMAX),alf(KMAXX,KMAXX),err(KMAXX),yerr(NMAX),
     *     ysav(NMAX),yseq(NMAX)
      LOGICAL first,reduct
      SAVE a,alf,epsold,first,kmax,kopt,nseq,xnew
      EXTERNAL derivs
      DATA first/.true./,epsold/-1./
      DATA nseq /2,4,6,8,10,12,14,16,18/
      if(eps.ne.epsold)then                    A new tolerance, so reinitialize.
        hnext=-1.e29                           "Impossible" values.
        xnew=-1.e29
        eps1=SAFE1*eps
        a(1)=nseq(1)+1                         Compute work coefficients A_k.
        do 11 k=1,KMAXX
          a(k+1)=a(k)+nseq(k+1)
        enddo 11
        do 13 iq=2,KMAXX                       Compute α(k,q).
          do 12 k=1,iq-1
            alf(k,iq)=eps1**((a(k+1)-a(iq+1))/
     *          ((a(iq+1)-a(1)+1.)*(2*k+1)))
          enddo 12
        enddo 13
        epsold=eps
        do 14 kopt=2,KMAXX-1                   Determine optimal row number for conver-
          if(a(kopt+1).gt.a(kopt)*alf(kopt-1,kopt))goto 1   gence.
        enddo 14
```

```
1       kmax=kopt
        endif
        h=htry
        do 15 i=1,nv                                  Save the starting values.
            ysav(i)=y(i)
        enddo 15
        if(h.ne.hnext.or.x.ne.xnew)then               A new stepsize or a new integration: re-establish
            first=.true.                                  the order window.
            kopt=kmax
        endif
        reduct=.false.
2       do 17 k=1,kmax                                Evaluate the sequence of modified midpoint
            xnew=x+h                                      integrations.
            if(xnew.eq.x)pause 'step size underflow in bsstep'
            call mmid(ysav,dydx,nv,x,h,nseq(k),yseq,derivs)
            xest=(h/nseq(k))**2                       Squared, since error series is even.
            call pzextr(k,xest,yseq,y,yerr,nv)        Perform extrapolation.
            if(k.ne.1)then                            Compute normalized error estimate ϵ(k).
                errmax=TINY
                do 16 i=1,nv
                    errmax=max(errmax,abs(yerr(i)/yscal(i)))
                enddo 16
                errmax=errmax/eps                     Scale error relative to tolerance.
                km=k-1
                err(km)=(errmax/SAFE1)**(1./(2*km+1))
            endif
            if(k.ne.1.and.(k.ge.kopt-1.or.first))then     In order window.
                if(errmax.lt.1.)goto 4                    Converged.
                if(k.eq.kmax.or.k.eq.kopt+1)then          Check for possible stepsize reduction.
                    red=SAFE2/err(km)
                    goto 3
                else if(k.eq.kopt)then
                    if(alf(kopt-1,kopt).lt.err(km))then
                        red=1./err(km)
                        goto 3
                    endif
                else if(kopt.eq.kmax)then
                    if(alf(km,kmax-1).lt.err(km))then
                        red=alf(km,kmax-1)*
*                            SAFE2/err(km)
                        goto 3
                    endif
                else if(alf(km,kopt).lt.err(km))then
                    red=alf(km,kopt-1)/err(km)
                    goto 3
                endif
            endif
        enddo 17
3       red=min(red,REDMIN)                           Reduce stepsize by at least REDMIN and at
        red=max(red,REDMAX)                               most REDMAX.
        h=h*red
        reduct=.true.
        goto 2                                        Try again.
4       x=xnew                                        Successful step taken.
        hdid=h
        first=.false.
        wrkmin=1.e35                                  Compute optimal row for convergence and
        do 18 kk=1,km                                     corresponding stepsize.
            fact=max(err(kk),SCALMX)
            work=fact*a(kk+1)
            if(work.lt.wrkmin)then
                scale=fact
                wrkmin=work
                kopt=kk+1
```

```
    endif
enddo 18
hnext=h/scale
if(kopt.ge.k.and.kopt.ne.kmax.and..not.reduct)then      Check for possible order increase, but not if stepsize was just reduced.
    fact=max(scale/alf(kopt-1,kopt),SCALMX)
    if(a(kopt+1)*fact.le.wrkmin)then
        hnext=h/fact
        kopt=kopt+1
    endif
endif
return
END
```

The polynomial extrapolation routine is based on the same algorithm as `polint` §3.1. It is simpler in that it is always extrapolating to zero, rather than to an arbitrary value. However, it is more complicated in that it must individually extrapolate each component of a vector of quantities.

```
SUBROUTINE pzextr(iest,xest,yest,yz,dy,nv)
INTEGER iest,nv,IMAX,NMAX
REAL xest,dy(nv),yest(nv),yz(nv)
PARAMETER (IMAX=13,NMAX=50)
    Use polynomial extrapolation to evaluate nv functions at x = 0 by fitting a polynomial to a
    sequence of estimates with progressively smaller values x = xest, and corresponding func-
    tion vectors yest(1:nv). This call is number iest in the sequence of calls. Extrapolated
    function values are output as yz(1:nv), and their estimated error is output as dy(1:nv).
    Parameters: Maximum expected value of iest is IMAX; of nv is NMAX.
INTEGER j,k1
REAL delta,f1,f2,q,d(NMAX),qcol(NMAX,IMAX),x(IMAX)
SAVE qcol,x
x(iest)=xest                    Save current independent variable.
do 11 j=1,nv
    dy(j)=yest(j)
    yz(j)=yest(j)
enddo 11
if(iest.eq.1) then              Store first estimate in first column.
    do 12 j=1,nv
        qcol(j,1)=yest(j)
    enddo 12
else
    do 13 j=1,nv
        d(j)=yest(j)
    enddo 13
    do 15 k1=1,iest-1
        delta=1./(x(iest-k1)-xest)
        f1=xest*delta
        f2=x(iest-k1)*delta
        do 14 j=1,nv            Propagate tableau 1 diagonal more.
            q=qcol(j,k1)
            qcol(j,k1)=dy(j)
            delta=d(j)-q
            dy(j)=f1*delta
            d(j)=f2*delta
            yz(j)=yz(j)+dy(j)
        enddo 14
    enddo 15
    do 16 j=1,nv
        qcol(j,iest)=dy(j)
    enddo 16
endif
return
END
```

Current wisdom favors polynomial extrapolation over rational function extrapolation in the Bulirsch-Stoer method. However, our feeling is that this view is guided more by the kinds of problems used for tests than by one method being actually "better." Accordingly, we provide the optional routine `rzextr` for rational function extrapolation, an exact substitution for `pzextr` above.

```
SUBROUTINE rzextr(iest,xest,yest,yz,dy,nv)
INTEGER iest,nv,IMAX,NMAX
REAL xest,dy(nv),yest(nv),yz(nv)
PARAMETER (IMAX=13,NMAX=50)
    Exact substitute for pzextr, but uses diagonal rational function extrapolation instead of
    polynomial extrapolation.
INTEGER j,k
REAL b,b1,c,ddy,v,yy,d(NMAX,IMAX),fx(IMAX),x(IMAX)
SAVE d,x
x(iest)=xest                    Save current independent variable.
if(iest.eq.1) then
    do 11 j=1,nv
        yz(j)=yest(j)
        d(j,1)=yest(j)
        dy(j)=yest(j)
    enddo 11
else
    do 12 k=1,iest-1
        fx(k+1)=x(iest-k)/xest
    enddo 12
    do 14 j=1,nv                Evaluate next diagonal in tableau.
        yy=yest(j)
        v=d(j,1)
        c=yy
        d(j,1)=yy
        do 13 k=2,iest
            b1=fx(k)*v
            b=b1-c
            if(b.ne.0.) then
                b=(c-v)/b
                ddy=c*b
                c=b1*b
            else                Care needed to avoid division by 0.
                ddy=v
            endif
            if (k.ne.iest) v=d(j,k)
            d(j,k)=ddy
            yy=yy+ddy
        enddo 13
        dy(j)=ddy
        yz(j)=yy
    enddo 14
endif
return
END
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §7.2.14. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422. [2]

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535. [3]

16.5 Second-Order Conservative Equations

Usually when you have a system of high-order differential equations to solve it is best to reformulate them as a system of first-order equations, as discussed in §16.0. There is a particular class of equations that occurs quite frequently in practice where you can gain about a factor of two in efficiency by differencing the equations directly. The equations are second-order systems where the derivative does not appear on the right-hand side:

$$y'' = f(x, y), \qquad y(x_0) = y_0, \qquad y'(x_0) = z_0 \tag{16.5.1}$$

As usual, y can denote a vector of values.

Stoermer's rule, dating back to 1907, has been a popular method for discretizing such systems. With $h = H/m$ we have

$$\begin{aligned} y_1 &= y_0 + h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_{k+1} - 2y_k + y_{k-1} &= h^2 f(x_0 + kh, y_k), \qquad k = 1, \ldots, m-1 \\ z_m &= (y_m - y_{m-1})/h + \tfrac{1}{2}hf(x_0 + H, y_m) \end{aligned} \tag{16.5.2}$$

Here z_m is $y'(x_0 + H)$. Henrici showed how to rewrite equations (16.5.2) to reduce roundoff error by using the quantities $\Delta_k \equiv y_{k+1} - y_k$. Start with

$$\begin{aligned} \Delta_0 &= h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_1 &= y_0 + \Delta_0 \end{aligned} \tag{16.5.3}$$

Then for $k = 1, \ldots, m-1$, set

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + h^2 f(x_0 + kh, y_k) \\ y_{k+1} &= y_k + \Delta_k \end{aligned} \tag{16.5.4}$$

Finally compute the derivative from

$$z_m = \Delta_{m-1}/h + \tfrac{1}{2}hf(x_0 + H, y_m) \tag{16.5.5}$$

Gragg again showed that the error series for equations (16.5.3)–(16.5.5) contains only even powers of h, and so the method is a logical candidate for extrapolation à la Bulirsch-Stoer. We replace `mmid` by the following routine `stoerm`:

```
      SUBROUTINE stoerm(y,d2y,nv,xs,htot,nstep,yout,derivs)
      INTEGER nstep,nv,NMAX
      REAL htot,xs,d2y(nv),y(nv),yout(nv)
      EXTERNAL derivs
      PARAMETER (NMAX=50)                    Maximum value of nv.
C     USES derivs
        Stoermer's rule for integrating y'' = f(x,y) for a system of n = nv/2 equations. On input
        y(1:nv) contains y in its first n elements and y' in its second n elements, all evaluated
        at xs. d2y(1:nv) contains the right-hand side function f (also evaluated at xs) in its
        first n elements. Its second n elements are not referenced. Also input is htot, the total
        step to be taken, and nstep, the number of substeps to be used. The output is returned
        as yout(1:nv), with the same storage arrangement as y. derivs is the user-supplied
        subroutine that calculates f.
      INTEGER i,n,neqns,nn
      REAL h,h2,halfh,x,ytemp(NMAX)
      h=htot/nstep                           Stepsize this trip.
      halfh=0.5*h
      neqns=nv/2                             Number of equations.
      do 11 i=1,neqns                        First step.
        n=neqns+i
        ytemp(n)=h*(y(n)+halfh*d2y(i))
```

16.5 Second-Order Conservative Equations

Usually when you have a system of high-order differential equations to solve it is best to reformulate them as a system of first-order equations, as discussed in §16.0. There is a particular class of equations that occurs quite frequently in practice where you can gain about a factor of two in efficiency by differencing the equations directly. The equations are second-order systems where the derivative does not appear on the right-hand side:

$$y'' = f(x, y), \qquad y(x_0) = y_0, \qquad y'(x_0) = z_0 \tag{16.5.1}$$

As usual, y can denote a vector of values.

Stoermer's rule, dating back to 1907, has been a popular method for discretizing such systems. With $h = H/m$ we have

$$\begin{aligned} y_1 &= y_0 + h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_{k+1} - 2y_k + y_{k-1} &= h^2 f(x_0 + kh, y_k), \qquad k = 1, \ldots, m-1 \\ z_m &= (y_m - y_{m-1})/h + \tfrac{1}{2}hf(x_0 + H, y_m) \end{aligned} \tag{16.5.2}$$

Here z_m is $y'(x_0 + H)$. Henrici showed how to rewrite equations (16.5.2) to reduce roundoff error by using the quantities $\Delta_k \equiv y_{k+1} - y_k$. Start with

$$\begin{aligned} \Delta_0 &= h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_1 &= y_0 + \Delta_0 \end{aligned} \tag{16.5.3}$$

Then for $k = 1, \ldots, m-1$, set

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + h^2 f(x_0 + kh, y_k) \\ y_{k+1} &= y_k + \Delta_k \end{aligned} \tag{16.5.4}$$

Finally compute the derivative from

$$z_m = \Delta_{m-1}/h + \tfrac{1}{2}hf(x_0 + H, y_m) \tag{16.5.5}$$

Gragg again showed that the error series for equations (16.5.3)–(16.5.5) contains only even powers of h, and so the method is a logical candidate for extrapolation à la Bulirsch-Stoer. We replace `mmid` by the following routine `stoerm`:

```
      SUBROUTINE stoerm(y,d2y,nv,xs,htot,nstep,yout,derivs)
      INTEGER nstep,nv,NMAX
      REAL htot,xs,d2y(nv),y(nv),yout(nv)
      EXTERNAL derivs
      PARAMETER (NMAX=50)                    Maximum value of nv.
C     USES derivs
        Stoermer's rule for integrating y'' = f(x,y) for a system of n = nv/2 equations. On input
        y(1:nv) contains y in its first n elements and y' in its second n elements, all evaluated
        at xs. d2y(1:nv) contains the right-hand side function f (also evaluated at xs) in its
        first n elements. Its second n elements are not referenced. Also input is htot, the total
        step to be taken, and nstep, the number of substeps to be used. The output is returned
        as yout(1:nv), with the same storage arrangement as y. derivs is the user-supplied
        subroutine that calculates f.
      INTEGER i,n,neqns,nn
      REAL h,h2,halfh,x,ytemp(NMAX)
      h=htot/nstep                           Stepsize this trip.
      halfh=0.5*h
      neqns=nv/2                             Number of equations.
      do 11 i=1,neqns                        First step.
        n=neqns+i
        ytemp(n)=h*(y(n)+halfh*d2y(i))
```

```
        ytemp(i)=y(i)+ytemp(n)
    enddo 11
    x=xs+h
    call derivs(x,ytemp,yout)          Use yout for temporary storage of derivatives.
    h2=h*h
    do 13 nn=2,nstep                   General step.
        do 12 i=1,neqns
            n=neqns+i
            ytemp(n)=ytemp(n)+h2*yout(i)
            ytemp(i)=ytemp(i)+ytemp(n)
        enddo 12
        x=x+h
        call derivs(x,ytemp,yout)
    enddo 13
    do 14 i=1,neqns                    Last step.
        n=neqns+i
        yout(n)=ytemp(n)/h+halfh*yout(i)
        yout(i)=ytemp(i)
    enddo 14
    return
    END
```

Note that for compatibility with `bsstep` the arrays `y` and `d2y` are of length $2n$ for a system of n second-order equations. The values of y are stored in the first n elements of `y`, while the first derivatives are stored in the second n elements. The right-hand side f is stored in the first n elements of the array `d2y`; the second n elements are unused. With this storage arrangement you can use `bsstep` simply by replacing the call to `mmid` with one to `stoerm` using the same arguments; just be sure that the argument `nv` of `bsstep` is set to $2n$. You should also use the more efficient sequence of stepsizes suggested by Deuflhard:

$$n = 1, 2, 3, 4, 5, \ldots \tag{16.5.6}$$

and set `KMAXX` $= 12$ in `bsstep`.

CITED REFERENCES AND FURTHER READING:

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

16.6 Stiff Sets of Equations

As soon as one deals with more than one first-order differential equation, the possibility of a *stiff* set of equations arises. Stiffness occurs in a problem where there are two or more very different scales of the independent variable on which the dependent variables are changing. For example, consider the following set of equations [1]:

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned} \tag{16.6.1}$$

with boundary conditions

$$u(0) = 1 \qquad v(0) = 0 \tag{16.6.2}$$

```
        ytemp(i)=y(i)+ytemp(n)
    enddo 11
    x=xs+h
    call derivs(x,ytemp,yout)          Use yout for temporary storage of derivatives.
    h2=h*h
    do 13 nn=2,nstep                   General step.
        do 12 i=1,neqns
            n=neqns+i
            ytemp(n)=ytemp(n)+h2*yout(i)
            ytemp(i)=ytemp(i)+ytemp(n)
        enddo 12
        x=x+h
        call derivs(x,ytemp,yout)
    enddo 13
    do 14 i=1,neqns                    Last step.
        n=neqns+i
        yout(n)=ytemp(n)/h+halfh*yout(i)
        yout(i)=ytemp(i)
    enddo 14
    return
    END
```

Note that for compatibility with `bsstep` the arrays `y` and `d2y` are of length $2n$ for a system of n second-order equations. The values of y are stored in the first n elements of `y`, while the first derivatives are stored in the second n elements. The right-hand side f is stored in the first n elements of the array `d2y`; the second n elements are unused. With this storage arrangement you can use `bsstep` simply by replacing the call to `mmid` with one to `stoerm` using the same arguments; just be sure that the argument `nv` of `bsstep` is set to $2n$. You should also use the more efficient sequence of stepsizes suggested by Deuflhard:

$$n = 1, 2, 3, 4, 5, \ldots \tag{16.5.6}$$

and set `KMAXX` $= 12$ in `bsstep`.

CITED REFERENCES AND FURTHER READING:

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

16.6 Stiff Sets of Equations

As soon as one deals with more than one first-order differential equation, the possibility of a *stiff* set of equations arises. Stiffness occurs in a problem where there are two or more very different scales of the independent variable on which the dependent variables are changing. For example, consider the following set of equations [1]:

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned} \tag{16.6.1}$$

with boundary conditions

$$u(0) = 1 \qquad v(0) = 0 \tag{16.6.2}$$

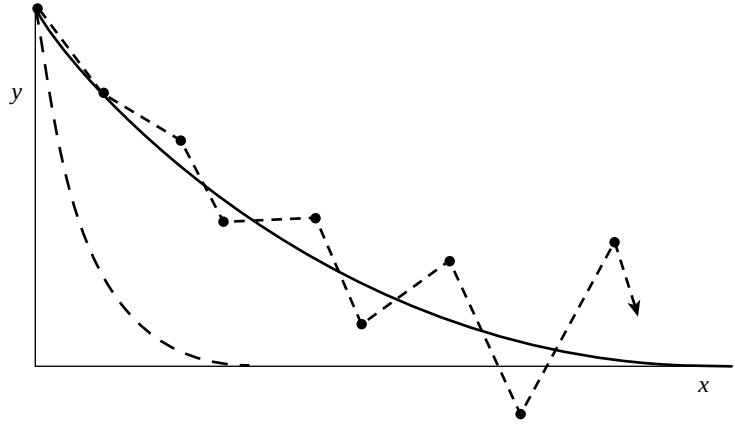


Figure 16.6.1. Example of an instability encountered in integrating a stiff equation (schematic). Here it is supposed that the equation has two solutions, shown as solid and dashed lines. Although the initial conditions are such as to give the solid solution, the stability of the integration (shown as the unstable dotted sequence of segments) is determined by the more rapidly varying dashed solution, even after that solution has effectively died away to zero. Implicit integration methods are the cure.

By means of the transformation

$$u = 2y - z \qquad v = -y + z \tag{16.6.3}$$

we find the solution

$$\begin{aligned} u &= 2e^{-x} - e^{-1000x} \\ v &= -e^{-x} + e^{-1000x} \end{aligned} \tag{16.6.4}$$

If we integrated the system (16.6.1) with any of the methods given so far in this chapter, the presence of the e^{-1000x} term would require a stepsize $h \ll 1/1000$ for the method to be stable (the reason for this is explained below). This is so even though the e^{-1000x} term is completely negligible in determining the values of u and v as soon as one is away from the origin (see Figure 16.6.1).

This is the generic disease of stiff equations: we are required to follow the variation in the solution on the shortest length scale to maintain stability of the integration, even though accuracy requirements allow a much larger stepsize.

To see how we might cure this problem, consider the single equation

$$y' = -cy \tag{16.6.5}$$

where $c > 0$ is a constant. The explicit (or *forward*) Euler scheme for integrating this equation with stepsize h is

$$y_{n+1} = y_n + hy'_n = (1 - ch)y_n \tag{16.6.6}$$

The method is called explicit because the new value y_{n+1} is given explicitly in terms of the old value y_n. Clearly the method is unstable if $h > 2/c$, for then $|y_n| \to \infty$ as $n \to \infty$.

The simplest cure is to resort to *implicit* differencing, where the right-hand side is evaluated at the *new* y location. In this case, we get the *backward Euler* scheme:

$$y_{n+1} = y_n + hy'_{n+1} \tag{16.6.7}$$

or

$$y_{n+1} = \frac{y_n}{1 + ch} \tag{16.6.8}$$

The method is absolutely stable: even as $h \to \infty$, $y_{n+1} \to 0$, which is in fact the correct solution of the differential equation. If we think of x as representing time, then the implicit method converges to the true equilibrium solution (i.e., the solution at late times) for large stepsizes. This nice feature of implicit methods holds only for linear systems, but even in the general case implicit methods give better stability. Of course, we give up *accuracy* in following the evolution towards equilibrium if we use large stepsizes, but we maintain *stability*.

These considerations can easily be generalized to sets of linear equations with constant coefficients:

$$\mathbf{y}' = -\mathbf{C} \cdot \mathbf{y} \tag{16.6.9}$$

where $\mathbf{C}$ is a positive definite matrix. Explicit differencing gives

$$\mathbf{y}_{n+1} = (\mathbf{1} - \mathbf{C}h) \cdot \mathbf{y}_n \tag{16.6.10}$$

Now a matrix $\mathbf{A}^n$ tends to zero as $n \to \infty$ only if the largest eigenvalue of $\mathbf{A}$ has magnitude less than unity. Thus $\mathbf{y}_n$ is bounded as $n \to \infty$ only if the largest eigenvalue of $\mathbf{1} - \mathbf{C}h$ is less than 1, or in other words

$$h < \frac{2}{\lambda_{\max}} \tag{16.6.11}$$

where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{C}$.

On the other hand, implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{y}'_{n+1} \tag{16.6.12}$$

or

$$\mathbf{y}_{n+1} = (\mathbf{1} + \mathbf{C}h)^{-1} \cdot \mathbf{y}_n \tag{16.6.13}$$

If the eigenvalues of $\mathbf{C}$ are λ, then the eigenvalues of $(\mathbf{1} + \mathbf{C}h)^{-1}$ are $(1 + \lambda h)^{-1}$, which has magnitude less than one for all h. (Recall that all the eigenvalues of a positive definite matrix are nonnegative.) Thus the method is stable for all stepsizes h. The penalty we pay for this stability is that we are required to invert a matrix at each step.

Not all equations are linear with constant coefficients, unfortunately! For the system

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}) \tag{16.6.14}$$

implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(\mathbf{y}_{n+1}) \tag{16.6.15}$$

In general this is some nasty set of nonlinear equations that has to be solved iteratively at each step. Suppose we try linearizing the equations, as in Newton's method:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{f}(\mathbf{y}_n) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right|_{\mathbf{y}_n} \cdot (\mathbf{y}_{n+1} - \mathbf{y}_n)\right] \tag{16.6.16}$$

Here $\partial \mathbf{f}/\partial \mathbf{y}$ is the matrix of the partial derivatives of the right-hand side (the Jacobian matrix). Rearrange equation (16.6.16) into the form

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot \mathbf{f}(\mathbf{y}_n) \tag{16.6.17}$$

If h is not too big, only one iteration of Newton's method may be accurate enough to solve equation (16.6.15) using equation (16.6.17). In other words, at each step we have to invert the matrix

$$\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}} \tag{16.6.18}$$

to find $\mathbf{y}_{n+1}$. Solving implicit methods by linearization is called a "semi-implicit" method, so equation (16.6.17) is the *semi-implicit Euler method*. It is not guaranteed to be stable, but it usually is, because the behavior is locally similar to the case of a constant matrix $\mathbf{C}$ described above.

So far we have dealt only with implicit methods that are first-order accurate. While these are very robust, most problems will benefit from higher-order methods. There are three important classes of higher-order methods for stiff systems:

- Generalizations of the Runge-Kutta method, of which the most useful are the Rosenbrock methods. The first practical implementation of these ideas was by Kaps and Rentrop, and so these methods are also called Kaps-Rentrop methods.
- Generalizations of the Bulirsch-Stoer method, in particular a semi-implicit extrapolation method due to Bader and Deuflhard.
- Predictor-corrector methods, most of which are descendants of Gear's backward differentiation method.

We shall give implementations of the first two methods. Note that systems where the right-hand side depends explicitly on x, $\mathbf{f}(\mathbf{y}, x)$, can be handled by adding x to the list of dependent variables so that the system to be solved is

$$\begin{pmatrix} \mathbf{y} \\ x \end{pmatrix}' = \begin{pmatrix} \mathbf{f} \\ 1 \end{pmatrix} \tag{16.6.19}$$

In both the routines to be given in this section, we have explicitly carried out this replacement for you, so the routines can handle right-hand sides of the form $\mathbf{f}(\mathbf{y}, x)$ without any special effort on your part.

We now mention an important point: *It is absolutely crucial to scale your variables properly when integrating stiff problems with automatic stepsize adjustment.* As in our nonstiff routines, you will be asked to supply a vector $\mathbf{y}_{\rm scal}$ with which the error is to be scaled. For example, to get constant fractional errors, simply set $\mathbf{y}_{\rm scal} = |\mathbf{y}|$. You can get constant absolute errors relative to some maximum values

by setting $\mathbf{y}_{\rm scal}$ equal to those maximum values. In stiff problems, there are often strongly decreasing pieces of the solution which you are not particularly interested in following once they are small. You can control the relative error above some threshold $\mathbf{C}$ and the absolute error below the threshold by setting

$$\mathbf{y}_{\rm scal} = \max(\mathbf{C}, |\mathbf{y}|) \tag{16.6.20}$$

If you are using appropriate nondimensional units, then each component of $\mathbf{C}$ should be of order unity. If you are not sure what values to take for $\mathbf{C}$, simply try setting each component equal to unity. *We strongly advocate the choice (16.6.20) for stiff problems.*

One final warning: Solving stiff problems can sometimes lead to catastrophic precision loss. Be alert for situations where double precision is necessary.

Rosenbrock Methods

These methods have the advantage of being relatively simple to understand and implement. For moderate accuracies ($\epsilon \lesssim 10^{-4} - 10^{-5}$ in the error criterion) and moderate-sized systems ($N \lesssim 10$), they are competitive with the more complicated algorithms. For more stringent parameters, Rosenbrock methods remain reliable; they merely become less efficient than competitors like the semi-implicit extrapolation method (see below).

A Rosenbrock method seeks a solution of the form

$$\mathbf{y}(x_0 + h) = \mathbf{y}_0 + \sum_{i=1}^{s} c_i \mathbf{k}_i \tag{16.6.21}$$

where the corrections $\mathbf{k}_i$ are found by solving s linear equations that generalize the structure in (16.6.17):

$$(\mathbf{1} - \gamma h \mathbf{f}') \cdot \mathbf{k}_i = h\mathbf{f}\left(\mathbf{y}_0 + \sum_{j=1}^{i-1} \alpha_{ij}\mathbf{k}_j\right) + h\mathbf{f}' \cdot \sum_{j=1}^{i-1} \gamma_{ij}\mathbf{k}_j, \qquad i = 1, \ldots, s \tag{16.6.22}$$

Here we denote the Jacobian matrix by $\mathbf{f}'$. The coefficients γ, c_i, α_{ij}, and γ_{ij} are fixed constants independent of the problem. If $\gamma = \gamma_{ij} = 0$, this is simply a Runge-Kutta scheme. Equations (16.6.22) can be solved successively for $\mathbf{k}_1, \mathbf{k}_2, \ldots$.

Crucial to the success of a stiff integration scheme is an automatic stepsize adjustment algorithm. Kaps and Rentrop [2] discovered an *embedded* or Runge-Kutta-Fehlberg method as described in §16.2: Two estimates of the form (16.6.21) are computed, the "real" one $\mathbf{y}$ and a lower-order estimate $\widehat{\mathbf{y}}$ with different coefficients $\hat{c}_i, i = 1, \ldots, \hat{s}$, where $\hat{s} < s$ but the $\mathbf{k}_i$ are the same. The difference between $\mathbf{y}$ and $\widehat{\mathbf{y}}$ leads to an estimate of the local truncation error, which can then be used for stepsize control. Kaps and Rentrop showed that the smallest value of s for which embedding is possible is $s = 4$, $\hat{s} = 3$, leading to a fourth-order method.

To minimize the matrix-vector multiplications on the right-hand side of (16.6.22), we rewrite the equations in terms of quantities

$$\mathbf{g}_i = \sum_{j=1}^{i-1} \gamma_{ij}\mathbf{k}_j + \gamma\mathbf{k}_i \tag{16.6.23}$$

The equations then take the form

$$\begin{aligned}
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_1 &= \mathbf{f}(\mathbf{y}_0) \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_2 &= \mathbf{f}(\mathbf{y}_0 + a_{21}\mathbf{g}_1) + c_{21}\mathbf{g}_1/h \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_3 &= \mathbf{f}(\mathbf{y}_0 + a_{31}\mathbf{g}_1 + a_{32}\mathbf{g}_2) + (c_{31}\mathbf{g}_1 + c_{32}\mathbf{g}_2)/h \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_4 &= \mathbf{f}(\mathbf{y}_0 + a_{41}\mathbf{g}_1 + a_{42}\mathbf{g}_2 + a_{43}\mathbf{g}_3) + (c_{41}\mathbf{g}_1 + c_{42}\mathbf{g}_2 + c_{43}\mathbf{g}_3)/h
\end{aligned} \tag{16.6.24}$$

In our implementation `stiff` of the Kaps-Rentrop algorithm, we have carried out the replacement (16.6.19) explicitly in equations (16.6.24), so you need not concern yourself about it. Simply provide a subroutine (called `derivs` in `stiff`) that returns **f** (called `dydx`) as a function of x and **y**. Also supply a subroutine `jacobn` that returns $\mathbf{f}'$ (`dfdy`) and $\partial\mathbf{f}/\partial x$ (`dfdx`) as functions of x and **y**. If x does not occur explicitly on the right-hand side, then `dfdx` will be zero. Usually the Jacobian matrix will be available to you by analytic differentiation of the right-hand side **f**. If not, your subroutine will have to compute it by numerical differencing with appropriate increments $\Delta\mathbf{y}$.

Kaps and Rentrop gave two different sets of parameters, which have slightly different stability properties. Several other sets have been proposed. Our default choice is that of Shampine [3], but we also give you one of the Kaps-Rentrop sets as an option. Some proposed parameter sets require function evaluations outside the domain of integration; we prefer to avoid that complication.

The calling sequence of `stiff` is exactly the same as the nonstiff routines given earlier in this chapter. It is thus "plug-compatible" with them in the general ODE integrating routine `odeint`. This compatibility requires, unfortunately, one slight anomaly: While the user-supplied routine `derivs` is a dummy argument (which can therefore have any actual name), the other user-supplied routine is *not* an argument and must be named (exactly) `jacobn`.

`stiff` begins by saving the initial values, in case the step has to be repeated because the error tolerance is exceeded. The linear equations (16.6.24) are solved by first computing the LU decomposition of the matrix $\mathbf{1}/\gamma h - \mathbf{f}'$ using the routine `ludcmp`. Then the four $\mathbf{g}_i$ are found by back-substitution of the four different right-hand sides using `lubksb`. Note that each step of the integration requires one call to `jacobn` and three calls to `derivs` (one call to get `dydx` before calling `stiff`, and two calls inside `stiff`). The reason only three calls are needed and not four is that the parameters have been chosen so that the last two calls in equation (16.6.24) are done with the same arguments. Counting the evaluation of the Jacobian matrix as roughly equivalent to N evaluations of the right-hand side **f**, we see that the Kaps-Rentrop scheme involves about $N + 3$ function evaluations per step. Note that if N is large and the Jacobian matrix is sparse, you should replace the LU decomposition by a suitable sparse matrix procedure.

Stepsize control depends on the fact that

$$\begin{aligned} \mathbf{y}_{\text{exact}} &= \mathbf{y} + O(h^5) \\ \mathbf{y}_{\text{exact}} &= \widehat{\mathbf{y}} + O(h^4) \end{aligned} \tag{16.6.25}$$

Thus

$$|\mathbf{y} - \widehat{\mathbf{y}}| = O(h^4) \tag{16.6.26}$$

Referring back to the steps leading from equation (16.2.4) to equation (16.2.10), we see that the new stepsize should be chosen as in equation (16.2.10) but with the exponents 1/4 and 1/5 replaced by 1/3 and 1/4, respectively. Also, experience shows that it is wise to prevent too large a stepsize change in one step, otherwise we will probably have to undo the large change in the next step. We adopt 0.5 and 1.5 as the maximum allowed decrease and increase of h in one step.

```
      SUBROUTINE stiff(y,dydx,n,x,htry,eps,yscal,hdid,hnext,derivs)
      INTEGER n,NMAX,MAXTRY
      REAL eps,hdid,hnext,htry,x,dydx(n),y(n),yscal(n),SAFETY,GROW,
     *     PGROW,SHRNK,PSHRNK,ERRCON,GAM,A21,A31,A32,A2X,A3X,C21,
     *     C31,C32,C41,C42,C43,B1,B2,B3,B4,E1,E2,E3,E4,C1X,C2X,C3X,
     *     C4X
      EXTERNAL derivs
      PARAMETER (NMAX=50,SAFETY=0.9,GROW=1.5,PGROW=-.25,
     *     SHRNK=0.5,PSHRNK=-1./3.,ERRCON=.1296,MAXTRY=40)
      PARAMETER (GAM=1./2.,A21=2.,A31=48./25.,A32=6./25.,C21=-8.,
     *     C31=372./25.,C32=12./5.,C41=-112./125.,C42=-54./125.,
     *     C43=-2./5.,B1=19./9.,B2=1./2.,B3=25./108.,B4=125./108.,
     *     E1=17./54.,E2=7./36.,E3=0.,E4=125./108.,C1X=1./2.,
```

```
*       C2X=-3./2.,C3X=121./50.,C4X=29./250.,A2X=1.,A3X=3./5.)
C   USES derivs,jacobn,lubksb,ludcmp
```

Fourth-order Rosenbrock step for integrating stiff o.d.e.'s, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y(1:n)` and its derivative `dydx(1:n)` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal(1:n)` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied subroutine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. Parameters: `NMAX` is the maximum value of `n`; `GROW` and `SHRNK` are the largest and smallest factors by which stepsize can change in one step; `ERRCON=(GROW/SAFETY)**(1/PGROW)` and handles the case when `errmax` $\simeq 0$.

```
      INTEGER i,j,jtry,indx(NMAX)
      REAL d,errmax,h,xsav,a(NMAX,NMAX),dfdx(NMAX),dfdy(NMAX,NMAX),
     *     dysav(NMAX),err(NMAX),g1(NMAX),g2(NMAX),g3(NMAX),
     *     g4(NMAX),ysav(NMAX)
      xsav=x                                  Save initial values.
      do 11 i=1,n
        ysav(i)=y(i)
        dysav(i)=dydx(i)
      enddo 11
      call jacobn(xsav,ysav,dfdx,dfdy,n,NMAX)
```

The user must supply this subroutine to return the `n`-by-`n` matrix `dfdy` and the vector `dfdx`.

```
      h=htry                                  Set stepsize to the initial trial value.
      do 23 jtry=1,MAXTRY
        do 13 i=1,n                           Set up the matrix 1 − γhf′.
          do 12 j=1,n
            a(i,j)=-dfdy(i,j)
          enddo 12
          a(i,i)=1./(GAM*h)+a(i,i)
        enddo 13
        call ludcmp(a,n,NMAX,indx,d)          LU decomposition of the matrix.
        do 14 i=1,n                           Set up right-hand side for g1.
          g1(i)=dysav(i)+h*C1X*dfdx(i)
        enddo 14
        call lubksb(a,n,NMAX,indx,g1)         Solve for g1.
        do 15 i=1,n                           Compute intermediate values of y and x.
          y(i)=ysav(i)+A21*g1(i)
        enddo 15
        x=xsav+A2X*h
        call derivs(x,y,dydx)                 Compute dydx at the intermediate values.
        do 16 i=1,n                           Set up right-hand side for g2.
          g2(i)=dydx(i)+h*C2X*dfdx(i)+C21*g1(i)/h
        enddo 16
        call lubksb(a,n,NMAX,indx,g2)         Solve for g2.
        do 17 i=1,n                           Compute intermediate values of y and x.
          y(i)=ysav(i)+A31*g1(i)+A32*g2(i)
        enddo 17
        x=xsav+A3X*h
        call derivs(x,y,dydx)                 Compute dydx at the intermediate values.
        do 18 i=1,n                           Set up right-hand side for g3.
          g3(i)=dydx(i)+h*C3X*dfdx(i)+(C31*g1(i)+
     *         C32*g2(i))/h
        enddo 18
        call lubksb(a,n,NMAX,indx,g3)         Solve for g3.
        do 19 i=1,n                           Set up right-hand side for g4.
          g4(i)=dydx(i)+h*C4X*dfdx(i)+(C41*g1(i)+
     *         C42*g2(i)+C43*g3(i))/h
        enddo 19
        call lubksb(a,n,NMAX,indx,g4)         Solve for g4.
        do 21 i=1,n                           Get fourth-order estimate of y and error estimate.
          y(i)=ysav(i)+B1*g1(i)+B2*g2(i)+B3*g3(i)+B4*g4(i)
```

```
            err(i)=E1*g1(i)+E2*g2(i)+E3*g3(i)+E4*g4(i)
        enddo 21
        x=xsav+h
        if(x.eq.xsav)pause 'stepsize not significant in stiff'
        errmax=0.                          Evaluate accuracy.
        do 22 i=1,n
            errmax=max(errmax,abs(err(i)/yscal(i)))
        enddo 22
        errmax=errmax/eps                  Scale relative to required tolerance.
        if(errmax.le.1.)then               Step succeeded. Compute size of next step and re-
            hdid=h                             turn.
            if(errmax.gt.ERRCON)then
                hnext=SAFETY*h*errmax**PGROW
            else
                hnext=GROW*h
            endif
            return
        else                               Truncation error too large, reduce stepsize.
            hnext=SAFETY*h*errmax**PSHRNK
            h=sign(max(abs(hnext),SHRNK*abs(h)),h)
        endif
    enddo 23                               Go back and re-try step.
    pause 'exceeded MAXTRY in stiff'
    END
```

Here are the Kaps-Rentrop parameters, which can be substituted for those of Shampine simply by replacing the `PARAMETER` statement:

```
      PARAMETER (GAM=.231,A21=2.,A31=4.52470820736,A32=4.16352878860,
     *     C21=-5.07167533877,C31=6.02015272865,C32=.159750684673,
     *     C41=-1.856343618677,C42=-8.50538085819,C43=
     *     -2.08407513602,B1=3.95750374663,B2=4.62489238836,B3=
     *     .617477263873,B4=1.282612945268,E1=-2.30215540292,
     *     E2=-3.07363448539,E3=.873280801802,E4=1.282612945268,
     *     C1X=GAM,C2X=-.396296677520e-01,C3X=.550778939579,
     *     C4X=-.553509845700e-01,A2X=.462,A3X=.880208333333)
```

As an example of how `stiff` is used, one can solve the system

$$\begin{aligned} y_1' &= -.013y_1 - 1000y_1y_3 \\ y_2' &= -2500y_2y_3 \\ y_3' &= -.013y_1 - 1000y_1y_3 - 2500y_2y_3 \end{aligned} \tag{16.6.27}$$

with initial conditions

$$y_1(0) = 1, \qquad y_2(0) = 1, \qquad y_3(0) = 0 \tag{16.6.28}$$

(This is test problem D4 in [4].) We integrate the system up to $x = 50$ with an initial stepsize of $h = 2.9 \times 10^{-4}$ using `odeint`. The components of $\mathbf{C}$ in (16.6.20) are all set to unity. The routines `derivs` and `jacobn` for this problem are given below. Even though the ratio of largest to smallest decay constants for this problem is around 10^6, `stiff` succeeds in integrating this set in only 29 steps with $\epsilon = 10^{-4}$. By contrast, the Runge-Kutta routine `rkqs` requires 51,012 steps!

```
      SUBROUTINE jacobn(x,y,dfdx,dfdy,n,nmax)
      INTEGER n,nmax,i
      REAL x,y(*),dfdx(*),dfdy(nmax,nmax)
      do 11 i=1,3
          dfdx(i)=0.
      enddo 11
```

```
dfdy(1,1)=-.013-1000.*y(3)
dfdy(1,2)=0.
dfdy(1,3)=-1000.*y(1)
dfdy(2,1)=0.
dfdy(2,2)=-2500.*y(3)
dfdy(2,3)=-2500.*y(2)
dfdy(3,1)=-.013-1000.*y(3)
dfdy(3,2)=-2500.*y(3)
dfdy(3,3)=-1000.*y(1)-2500.*y(2)
return
END

SUBROUTINE derivs(x,y,dydx)
REAL x,y(*),dydx(*)
dydx(1)=-.013*y(1)-1000.*y(1)*y(3)
dydx(2)=-2500.*y(2)*y(3)
dydx(3)=-.013*y(1)-1000.*y(1)*y(3)-2500.*y(2)*y(3)
return
END
```

Semi-implicit Extrapolation Method

The Bulirsch-Stoer method, which discretizes the differential equation using the modified midpoint rule, does not work for stiff problems. Bader and Deuflhard [5] discovered a semi-implicit discretization that works very well and that lends itself to extrapolation exactly as in the original Bulirsch-Stoer method.

The starting point is an implicit form of the midpoint rule:

$$\mathbf{y}_{n+1} - \mathbf{y}_{n-1} = 2h\mathbf{f}\left(\frac{\mathbf{y}_{n+1} + \mathbf{y}_{n-1}}{2}\right) \tag{16.6.29}$$

Convert this equation into semi-implicit form by linearizing the right-hand side about $\mathbf{f}(\mathbf{y}_n)$. The result is the *semi-implicit midpoint rule*:

$$\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n+1} = \left[\mathbf{1} + h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n-1} + 2h\left[\mathbf{f}(\mathbf{y}_n) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \cdot \mathbf{y}_n\right] \tag{16.6.30}$$

It is used with a special first step, the semi-implicit Euler step (16.6.17), and a special "smoothing" last step in which the last $\mathbf{y}_n$ is replaced by

$$\overline{\mathbf{y}}_n \equiv \tfrac{1}{2}(\mathbf{y}_{n+1} + \mathbf{y}_{n-1}) \tag{16.6.31}$$

Bader and Deuflhard showed that the error series for this method once again involves only even powers of h.

For practical implementation, it is better to rewrite the equations using $\Delta_k \equiv \mathbf{y}_{k+1} - \mathbf{y}_k$. With $h = H/m$, start by calculating

$$\begin{aligned} \Delta_0 &= \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot h\mathbf{f}(\mathbf{y}_0) \\ \mathbf{y}_1 &= \mathbf{y}_0 + \Delta_0 \end{aligned} \tag{16.6.32}$$

Then for $k = 1, \ldots, m-1$, set

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + 2\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1}] \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + \Delta_k \end{aligned} \tag{16.6.33}$$

Finally compute

$$\Delta_m = \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}]$$
$$\overline{\mathbf{y}}_m = \mathbf{y}_m + \Delta_m \tag{16.6.34}$$

It is easy to incorporate the replacement (16.6.19) in the above formulas. The additional terms in the Jacobian that come from $\partial \mathbf{f}/\partial x$ all cancel out of the semi-implicit midpoint rule (16.6.30). In the special first step (16.6.17), and in the corresponding equation (16.6.32), the term $h\mathbf{f}$ becomes $h\mathbf{f} + h^2\partial \mathbf{f}/\partial x$. The remaining equations are all unchanged.

This algorithm is implemented in the routine `simpr`:

```
      SUBROUTINE simpr(y,dydx,dfdx,dfdy,nmax,n,xs,htot,nstep,yout,
     *     derivs)
      INTEGER n,nmax,nstep,NMAXX
      REAL htot,xs,dfdx(n),dfdy(nmax,nmax),dydx(n),y(n),yout(n)
      EXTERNAL derivs
      PARAMETER (NMAXX=50)                    Maximum expected value of n.
C     USES derivs,lubksb,ludcmp
        Performs one step of semi-implicit midpoint rule. Input are the dependent variable y(1:n),
        its derivative dydx(1:n), the derivative of the right-hand side with respect to x, dfdx(1:n),
        and the Jacobian dfdy(1:nmax,1:nmax) at xs. Also input are htot, the total step
        to be taken, and nstep, the number of substeps to be used. The output is returned as
        yout(1:n). derivs is the user-supplied subroutine that calculates dydx.
      INTEGER i,j,nn,indx(NMAXX)
      REAL d,h,x,a(NMAXX,NMAXX),del(NMAXX),ytemp(NMAXX)
      h=htot/nstep                            Stepsize this trip.
      do 12 i=1,n                             Set up the matrix 1 − hf′.
        do 11 j=1,n
          a(i,j)=-h*dfdy(i,j)
        enddo 11
        a(i,i)=a(i,i)+1.
      enddo 12
      call ludcmp(a,n,NMAXX,indx,d)           LU decomposition of the matrix.
      do 13 i=1,n                             Set up right-hand side for first step. Use yout for
        yout(i)=h*(dydx(i)+h*dfdx(i))           temporary storage.
      enddo 13
      call lubksb(a,n,NMAXX,indx,yout)
      do 14 i=1,n                             First step.
        del(i)=yout(i)
        ytemp(i)=y(i)+del(i)
      enddo 14
      x=xs+h
      call derivs(x,ytemp,yout)               Use yout for temporary storage of derivatives.
      do 17 nn=2,nstep                        General step.
        do 15 i=1,n                           Set up right-hand side for general step.
          yout(i)=h*yout(i)-del(i)
        enddo 15
        call lubksb(a,n,NMAXX,indx,yout)
        do 16 i=1,n
          del(i)=del(i)+2.*yout(i)
          ytemp(i)=ytemp(i)+del(i)
        enddo 16
        x=x+h
        call derivs(x,ytemp,yout)
      enddo 17
      do 18 i=1,n                             Set up right-hand side for last step.
        yout(i)=h*yout(i)-del(i)
      enddo 18
      call lubksb(a,n,NMAXX,indx,yout)
      do 19 i=1,n                             Take last step.
        yout(i)=ytemp(i)+yout(i)
```

```
      enddo 19
      return
      END
```

The routine simpr is intended to be used in a routine stifbs that is almost exactly the same as bsstep. The only differences are:

- The stepsize sequence is

$$n = 2, 6, 10, 14, 22, 34, 50, \ldots, \tag{16.6.35}$$

  where each member differs from its predecessor by the smallest multiple of 4 that makes the ratio of successive terms be $\leq \frac{5}{7}$. The parameter KMAXX is taken to be 7.
- The work per unit step now includes the cost of Jacobian evaluations as well as function evaluations. We count one Jacobian evaluation as equivalent to N function evaluations, where N is the number of equations.
- Once again the user-supplied routine derivs is a dummy argument and so can have any name. However, to maintain "plug-compatibility" with rkqs, bsstep and stiff, the routine jacobn is not an argument and *must* have exactly this name. It is called once per step to return $\mathbf{f}'$ (dfdy) and $\partial\mathbf{f}/\partial x$ (dfdx) as functions of x and $\mathbf{y}$.

Here is the routine, with comments pointing out only the differences from bsstep:

```
      SUBROUTINE stifbs(y,dydx,nv,x,htry,eps,yscal,hdid,hnext,derivs)
      INTEGER nv,NMAX,KMAXX,IMAX
      REAL eps,hdid,hnext,htry,x,dydx(nv),y(nv),yscal(nv),SAFE1,
     *     SAFE2,REDMAX,REDMIN,TINY,SCALMX
      EXTERNAL derivs
      PARAMETER (NMAX=50,KMAXX=7,IMAX=KMAXX+1,SAFE1=.25,SAFE2=.7,
     *     REDMAX=1.e-5,REDMIN=.7,TINY=1.e-30,SCALMX=.1)
C     USES derivs,jacobn,simpr,pzextr
```

Semi-implicit extrapolation step for integrating stiff o.d.e.'s, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector y(1:n) and its derivative dydx(1:n) at the starting value of the independent variable x. Also input are the stepsize to be attempted htry, the required accuracy eps, and the vector yscal(1:n) against which the error is scaled. On output, y and x are replaced by their new values, hdid is the stepsize that was actually accomplished, and hnext is the estimated next stepsize. derivs is a user-supplied subroutine that computes the derivatives of the right-hand side with respect to x, while jacobn (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of y. Be sure to set htry on successive steps to the value of hnext returned from the previous step, as is the case if the routine is called by odeint.

```
      INTEGER i,iq,k,kk,km,kmax,kopt,nvold,nseq(IMAX)
      REAL eps1,epsold,errmax,fact,h,red,scale,work,wrkmin,xest,xnew,
     *     a(IMAX),alf(KMAXX,KMAXX),dfdx(NMAX),dfdy(NMAX,NMAX),
     *     err(KMAXX),yerr(NMAX),ysav(NMAX),yseq(NMAX)
      LOGICAL first,reduct
      SAVE a,alf,epsold,first,kmax,kopt,nseq,nvold,xnew
      DATA first/.true./,epsold/-1./,nvold/-1/
      DATA nseq /2,6,10,14,22,34,50,70/          Sequence is different from bsstep.
      if(eps.ne.epsold.or.nv.ne.nvold)then       Reinitialize also if nv has changed.
        hnext=-1.e29
        xnew=-1.e29
        eps1=SAFE1*eps
        a(1)=nseq(1)+1
        do 11 k=1,KMAXX
          a(k+1)=a(k)+nseq(k+1)
        enddo 11
        do 13 iq=2,KMAXX
          do 12 k=1,iq-1
            alf(k,iq)=eps1**((a(k+1)-a(iq+1))/
     *           ((a(iq+1)-a(1)+1.)*(2*k+1)))
          enddo 12
        enddo 13
```

```
        epsold=eps
        nvold=nv                                  Save nv.
        a(1)=nv+a(1)                              Add cost of Jacobian evaluations to work co-
        do 14 k=1,KMAXX                               efficients.
            a(k+1)=a(k)+nseq(k+1)
        enddo 14
        do 15 kopt=2,KMAXX-1
            if(a(kopt+1).gt.a(kopt)*alf(kopt-1,kopt))goto 1
        enddo 15
1       kmax=kopt
    endif
    h=htry
    do 16 i=1,nv
        ysav(i)=y(i)
    enddo 16
    call jacobn(x,y,dfdx,dfdy,nv,nmax)            Evaluate Jacobian.
    if(h.ne.hnext.or.x.ne.xnew)then
        first=.true.
        kopt=kmax
    endif
    reduct=.false.
2   do 18 k=1,kmax
        xnew=x+h
        if(xnew.eq.x)pause 'stepsize underflow in stifbs'
        call simpr(ysav,dydx,dfdx,dfdy,nmax,nv,x,h,nseq(k),yseq,
*            derivs)                               Semi-implicit midpoint rule.
        xest=(h/nseq(k))**2                       The rest of the routine is identical to bsstep.
        call pzextr(k,xest,yseq,y,yerr,nv)
        if(k.ne.1)then
            errmax=TINY
            do 17 i=1,nv
                errmax=max(errmax,abs(yerr(i)/yscal(i)))
            enddo 17
            errmax=errmax/eps
            km=k-1
            err(km)=(errmax/SAFE1)**(1./(2*km+1))
        endif
        if(k.ne.1.and.(k.ge.kopt-1.or.first))then
            if(errmax.lt.1.)goto 4
            if(k.eq.kmax.or.k.eq.kopt+1)then
                red=SAFE2/err(km)
                goto 3
            else if(k.eq.kopt)then
                if(alf(kopt-1,kopt).lt.err(km))then
                    red=1./err(km)
                    goto 3
                endif
            else if(kopt.eq.kmax)then
                if(alf(km,kmax-1).lt.err(km))then
                    red=alf(km,kmax-1)*
*                       SAFE2/err(km)
                    goto 3
                endif
            else if(alf(km,kopt).lt.err(km))then
                red=alf(km,kopt-1)/err(km)
                goto 3
            endif
        endif
    enddo 18
3   red=min(red,REDMIN)
    red=max(red,REDMAX)
    h=h*red
    reduct=.true.
    goto 2
```

```
4   x=xnew
    hdid=h
    first=.false.
    wrkmin=1.e35
    do 19 kk=1,km
        fact=max(err(kk),SCALMX)
        work=fact*a(kk+1)
        if(work.lt.wrkmin)then
            scale=fact
            wrkmin=work
            kopt=kk+1
        endif
    enddo 19
    hnext=h/scale
    if(kopt.ge.k.and.kopt.ne.kmax.and..not.reduct)then
        fact=max(scale/alf(kopt-1,kopt),SCALMX)
        if(a(kopt+1)*fact.le.wrkmin)then
            hnext=h/fact
            kopt=kopt+1
        endif
    endif
    return
    END
```

The routine `stifbs` is an excellent routine for all stiff problems, competitive with the best Gear-type routines. `stiff` is comparable in execution time for moderate N and $\epsilon \lesssim 10^{-4}$. By the time $\epsilon \sim 10^{-8}$, `stifbs` is roughly an order of magnitude faster. There are further improvements that could be applied to `stifbs` to make it even more robust. For example, very occasionally `ludcmp` in `simpr` will encounter a singular matrix. You could arrange for the stepsize to be reduced, say by a factor of the current `nseq(k)`. There are also certain stability restrictions on the stepsize that come into play on some problems. For a discussion of how to implement these automatically, see [6].

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Kaps, P., and Rentrop, P. 1979, *Numerische Mathematik*, vol. 33, pp. 55–68. [2]

Shampine, L.F. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 93–113. [3]

Enright, W.H., and Pryce, J.D. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 1–27. [4]

Bader, G., and Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 373–398. [5]

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422.

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

Deuflhard, P. 1987, "Uniqueness Theorems for Stiff ODE Initial Value Problems," *Preprint SC-87-3* (Berlin: Konrad Zuse Zentrum für Informationstechnik). [6]

Enright, W.H., Hull, T.E., and Lindberg, B. 1975, *BIT*, vol. 15, pp. 10–48.

Wanner, G. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics, vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag).

16.7 Multistep, Multivalue, and Predictor-Corrector Methods

The terms multistep and multivalue describe two different ways of implementing essentially the same integration technique for ODEs. Predictor-corrector is a particular subcategrory of these methods — in fact, the most widely used. Accordingly, the name predictor-corrector is often loosely used to denote all these methods.

We suspect that predictor-corrector integrators have had their day, and that they are no longer the method of choice for most problems in ODEs. For high-precision applications, or applications where evaluations of the right-hand sides are expensive, Bulirsch-Stoer dominates. For convenience, or for low precision, adaptive-stepsize Runge-Kutta dominates. Predictor-corrector methods have been, we think, squeezed out in the middle. There is possibly only one exceptional case: high-precision solution of very smooth equations with very complicated right-hand sides, as we will describe later.

Nevertheless, these methods have had a long historical run. Textbooks are full of information on them, and there are a lot of standard ODE programs around that are based on predictor-corrector methods. Many capable researchers have a lot of experience with predictor-corrector routines, and they see no reason to make a precipitous change of habit. It is not a bad idea for you to be familiar with the principles involved, and even with the sorts of bookkeeping details that are the bane of these methods. Otherwise there will be a big surprise in store when you first have to fix a problem in a predictor-corrector routine.

Let us first consider the multistep approach. Think about how integrating an ODE is different from finding the integral of a function: For a function, the integrand has a known dependence on the independent variable x, and can be evaluated at will. For an ODE, the "integrand" is the right-hand side, which depends both on x and on the dependent variables y. Thus to advance the solution of $y' = f(x, y)$ from x_n to x, we have

$$y(x) = y_n + \int_{x_n}^{x} f(x', y)\, dx' \tag{16.7.1}$$

In a single-step method like Runge-Kutta or Bulirsch-Stoer, the value y_{n+1} at x_{n+1} depends only on y_n. In a multistep method, we approximate $f(x, y)$ by a polynomial passing through *several* previous points $x_n, x_{n-1}, \ldots$ and possibly also through x_{n+1}. The result of evaluating the integral (16.7.1) at $x = x_{n+1}$ is then of the form

$$y_{n+1} = y_n + h(\beta_0 y'_{n+1} + \beta_1 y'_n + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \cdots) \tag{16.7.2}$$

where y'_n denotes $f(x_n, y_n)$, and so on. If $\beta_0 = 0$, the method is explicit; otherwise it is implicit. The order of the method depends on how many previous steps we use to get each new value of y.

Consider how we might solve an implicit formula of the form (16.7.2) for y_{n+1}. Two methods suggest themselves: *functional iteration* and *Newton's method*. In functional iteration, we take some initial guess for y_{n+1}, insert it into the right-hand side of (16.7.2) to get an updated value of y_{n+1}, insert this updated value back into the right-hand side, and continue iterating. But how are we to get an initial guess for

16.7 Multistep, Multivalue, and Predictor-Corrector Methods

The terms multistep and multivalue describe two different ways of implementing essentially the same integration technique for ODEs. Predictor-corrector is a particular subcategrory of these methods — in fact, the most widely used. Accordingly, the name predictor-corrector is often loosely used to denote all these methods.

We suspect that predictor-corrector integrators have had their day, and that they are no longer the method of choice for most problems in ODEs. For high-precision applications, or applications where evaluations of the right-hand sides are expensive, Bulirsch-Stoer dominates. For convenience, or for low precision, adaptive-stepsize Runge-Kutta dominates. Predictor-corrector methods have been, we think, squeezed out in the middle. There is possibly only one exceptional case: high-precision solution of very smooth equations with very complicated right-hand sides, as we will describe later.

Nevertheless, these methods have had a long historical run. Textbooks are full of information on them, and there are a lot of standard ODE programs around that are based on predictor-corrector methods. Many capable researchers have a lot of experience with predictor-corrector routines, and they see no reason to make a precipitous change of habit. It is not a bad idea for you to be familiar with the principles involved, and even with the sorts of bookkeeping details that are the bane of these methods. Otherwise there will be a big surprise in store when you first have to fix a problem in a predictor-corrector routine.

Let us first consider the multistep approach. Think about how integrating an ODE is different from finding the integral of a function: For a function, the integrand has a known dependence on the independent variable x, and can be evaluated at will. For an ODE, the "integrand" is the right-hand side, which depends both on x and on the dependent variables y. Thus to advance the solution of $y' = f(x, y)$ from x_n to x, we have

$$y(x) = y_n + \int_{x_n}^{x} f(x', y)\, dx' \tag{16.7.1}$$

In a single-step method like Runge-Kutta or Bulirsch-Stoer, the value y_{n+1} at x_{n+1} depends only on y_n. In a multistep method, we approximate $f(x, y)$ by a polynomial passing through *several* previous points $x_n, x_{n-1}, \ldots$ and possibly also through x_{n+1}. The result of evaluating the integral (16.7.1) at $x = x_{n+1}$ is then of the form

$$y_{n+1} = y_n + h(\beta_0 y'_{n+1} + \beta_1 y'_n + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \cdots) \tag{16.7.2}$$

where y'_n denotes $f(x_n, y_n)$, and so on. If $\beta_0 = 0$, the method is explicit; otherwise it is implicit. The order of the method depends on how many previous steps we use to get each new value of y.

Consider how we might solve an implicit formula of the form (16.7.2) for y_{n+1}. Two methods suggest themselves: *functional iteration* and *Newton's method*. In functional iteration, we take some initial guess for y_{n+1}, insert it into the right-hand side of (16.7.2) to get an updated value of y_{n+1}, insert this updated value back into the right-hand side, and continue iterating. But how are we to get an initial guess for

y_{n+1}? Easy! Just use some *explicit* formula of the same form as (16.7.2). This is called the *predictor step*. In the predictor step we are essentially *extrapolating* the polynomial fit to the derivative from the previous points to the new point x_{n+1} and then doing the integral (16.7.1) in a Simpson-like manner from x_n to x_{n+1}. The subsequent Simpson-like integration, using the prediction step's value of y_{n+1} to *interpolate* the derivative, is called the *corrector step*. The difference between the predicted and corrected function values supplies information on the local truncation error that can be used to control accuracy and to adjust stepsize.

If one corrector step is good, aren't many better? Why not use each corrector as an improved predictor and iterate to convergence on each step? Answer: Even if you had a *perfect* predictor, the step would still be accurate only to the finite order of the corrector. This incurable error term is on the same order as that which your iteration is supposed to cure, so you are at best changing only the coefficient in front of the error term by a fractional amount. So dubious an improvement is certainly not worth the effort. Your extra effort would be better spent in taking a smaller stepsize.

As described so far, you might think it desirable or necessary to predict several intervals ahead at each step, then to use all these intervals, with various weights, in a Simpson-like corrector step. That is not a good idea. Extrapolation is the least stable part of the procedure, and it is desirable to minimize its effect. Therefore, the integration steps of a predictor-corrector method are overlapping, each one involving several stepsize intervals h, but extending just one such interval farther than the previous ones. Only that one extended interval is extrapolated by each predictor step.

The most popular predictor-corrector methods are probably the Adams-Bashforth-Moulton schemes, which have good stability properties. The Adams-Bashforth part is the predictor. For example, the third-order case is

$$\text{predictor:} \qquad y_{n+1} = y_n + \frac{h}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + O(h^4) \tag{16.7.3}$$

Here information at the current point x_n, together with the two previous points x_{n-1} and x_{n-2} (assumed equally spaced), is used to predict the value y_{n+1} at the next point, x_{n+1}. The Adams-Moulton part is the corrector. The third-order case is

$$\text{corrector:} \qquad y_{n+1} = y_n + \frac{h}{12}(5y'_{n+1} + 8y'_n - y'_{n-1}) + O(h^4) \tag{16.7.4}$$

Without the trial value of y_{n+1} from the predictor step to insert on the right-hand side, the corrector would be a nasty implicit equation for y_{n+1}.

There are actually three separate processes occurring in a predictor-corrector method: the predictor step, which we call P, the evaluation of the derivative y'_{n+1} from the latest value of y, which we call E, and the corrector step, which we call C. In this notation, iterating m times with the corrector (a practice we inveighed against earlier) would be written P(EC)^m. One also has the choice of finishing with a C or an E step. The lore is that a final E is superior, so the strategy usually recommended is PECE.

Notice that a PC method with a fixed number of iterations (say, one) is an explicit method! When we fix the number of iterations in advance, then the final value of y_{n+1} can be written as some complicated function of known quantities. Thus fixed iteration PC methods lose the strong stability properties of implicit methods and *should only be used for nonstiff problems.*

For stiff problems we *must* use an implicit method if we want to avoid having tiny stepsizes. (Not all implicit methods are good for stiff problems, but fortunately some good ones such as the Gear formulas are known.) We then appear to have two choices for solving the implicit equations: functional iteration to convergence, or Newton iteration. However, it turns out that for stiff problems functional iteration will not even converge unless we use tiny stepsizes, no matter how close our prediction is! Thus Newton iteration is usually an essential part of a multistep stiff solver. For convergence, Newton's method doesn't particularly care what the stepsize is, as long as the prediction is accurate enough.

Multistep methods, as we have described them so far, suffer from two serious difficulties when one tries to implement them:

- Since the formulas require results from equally spaced steps, adjusting the stepsize is difficult.
- Starting and stopping present problems. For starting, we need the initial values plus several previous steps to prime the pump. Stopping is a problem because equal steps are unlikely to land directly on the desired termination point.

Older implementations of PC methods have various cumbersome ways of dealing with these problems. For example, they might use Runge-Kutta to start and stop. Changing the stepsize requires considerable bookkeeping to do some kind of interpolation procedure. Fortunately both these drawbacks disappear with the multivalue approach.

For multivalue methods the basic data available to the integrator are the first few terms of the Taylor series expansion of the solution at the current point x_n. The aim is to advance the solution and obtain the expansion coefficients at the next point x_{n+1}. This is in contrast to multistep methods, where the data are the values of the solution at $x_n, x_{n-1}, \ldots$. We'll illustrate the idea by considering a four-value method, for which the basic data are

$$\mathbf{y}_n \equiv \begin{pmatrix} y_n \\ hy'_n \\ (h^2/2)y''_n \\ (h^3/6)y'''_n \end{pmatrix} \tag{16.7.5}$$

It is also conventional to scale the derivatives with the powers of $h = x_{n+1} - x_n$ as shown. Note that here we use the vector notation $\mathbf{y}$ to denote the solution and its first few derivatives at a point, not the fact that we are solving a system of equations with many components y.

In terms of the data in (16.7.5), we can approximate the value of the solution y at some point x:

$$y(x) = y_n + (x - x_n)y'_n + \frac{(x - x_n)^2}{2}y''_n + \frac{(x - x_n)^3}{6}y'''_n \tag{16.7.6}$$

Set $x = x_{n+1}$ in equation (16.7.6) to get an approximation to y_{n+1}. Differentiate equation (16.7.6) and set $x = x_{n+1}$ to get an approximation to y'_{n+1}, and similarly for y''_{n+1} and y'''_{n+1}. Call the resulting approximation $\tilde{\mathbf{y}}_{n+1}$, where the tilde is a reminder

that all we have done so far is a polynomial extrapolation of the solution and its derivatives; we have not yet used the differential equation. You can easily verify that

$$\widetilde{\mathbf{y}}_{n+1} = \mathbf{B} \cdot \mathbf{y}_n \tag{16.7.7}$$

where the matrix $\mathbf{B}$ is

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{16.7.8}$$

We now write the actual approximation to $\mathbf{y}_{n+1}$ that we will use by adding a correction to $\widetilde{\mathbf{y}}_{n+1}$:

$$\mathbf{y}_{n+1} = \widetilde{\mathbf{y}}_{n+1} + \alpha \mathbf{r} \tag{16.7.9}$$

Here $\mathbf{r}$ will be a fixed vector of numbers, in the same way that $\mathbf{B}$ is a fixed matrix. We fix α by requiring that the differential equation

$$y'_{n+1} = f(x_{n+1}, y_{n+1}) \tag{16.7.10}$$

be satisfied. The second of the equations in (16.7.9) is

$$hy'_{n+1} = h\widetilde{y}'_{n+1} + \alpha r_2 \tag{16.7.11}$$

and this will be consistent with (16.7.10) provided

$$r_2 = 1, \qquad \alpha = hf(x_{n+1}, y_{n+1}) - h\widetilde{y}'_{n+1} \tag{16.7.12}$$

The values of r_1, r_3, and r_4 are free for the inventor of a given four-value method to choose. Different choices give different orders of method (i.e., through what order in h the final expression 16.7.9 actually approximates the solution), and different stability properties.

An interesting result, not obvious from our presentation, is that multivalue and multistep methods are entirely equivalent. In other words, the value y_{n+1} given by a multivalue method with given $\mathbf{B}$ and $\mathbf{r}$ is exactly the same value given by some multistep method with given β's in equation (16.7.2). For example, it turns out that the Adams-Bashforth formula (16.7.3) corresponds to a four-value method with $r_1 = 0$, $r_3 = 3/4$, and $r_4 = 1/6$. The method is explicit because $r_1 = 0$. The Adams-Moulton method (16.7.4) corresponds to the implicit four-value method with $r_1 = 5/12$, $r_3 = 3/4$, and $r_4 = 1/6$. Implicit multivalue methods are solved the same way as implicit multistep methods: either by a predictor-corrector approach using an explicit method for the predictor, or by Newton iteration for stiff systems.

Why go to all the trouble of introducing a whole new method that turns out to be equivalent to a method you already knew? The reason is that multivalue methods allow an easy solution to the two difficulties we mentioned above in actually implementing multistep methods.

Consider first the question of stepsize adjustment. To change stepsize from h to h' at some point x_n, simply multiply the components of $\mathbf{y}_n$ in (16.7.5) by the appropriate powers of h'/h, and you are ready to continue to $x_n + h'$.

Multivalue methods also allow a relatively easy change in the *order* of the method: Simply change **r**. The usual strategy for this is first to determine the new stepsize with the current order from the error estimate. Then check what stepsize would be predicted using an order one greater and one smaller than the current order. Choose the order that allows you to take the biggest next step. Being able to change order also allows an easy solution to the starting problem: Simply start with a first-order method and let the order automatically increase to the appropriate level.

For low accuracy requirements, a Runge-Kutta routine like `rkqs` is almost always the most efficient choice. For high accuracy, `bsstep` is both robust and efficient. For very smooth functions, a variable-order PC method can invoke very high orders. If the right-hand side of the equation is relatively complicated, so that the expense of evaluating it outweighs the bookkeeping expense, then the best PC packages can outperform Bulirsch-Stoer on such problems. As you can imagine, however, such a variable-stepsize, variable-order method is not trivial to program. If you suspect that your problem is suitable for this treatment, we recommend use of a canned PC package. For further details consult Gear [1] or Shampine and Gordon [2].

Our prediction, nevertheless, is that, as extrapolation methods like Bulirsch-Stoer continue to gain sophistication, they will eventually beat out PC methods in all applications. We are willing, however, to be corrected.

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Shampine, L.F., and Gordon, M.K. 1975, *Computer Solution of Ordinary Differential Equations. The Initial Value Problem.* (San Francisco: W.H Freeman). [2]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 8.

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists*; reprinted 1986 (New York: Dover), Chapters 14–15.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.

Chapter 17. Two Point Boundary Value Problems

17.0 Introduction

When ordinary differential equations are required to satisfy boundary conditions at more than one value of the independent variable, the resulting problem is called a *two point boundary value problem*. As the terminology indicates, the most common case by far is where boundary conditions are supposed to be satisfied at two points — usually the starting and ending values of the integration. However, the phrase "two point boundary value problem" is also used loosely to include more complicated cases, e.g., where some conditions are specified at endpoints, others at interior (usually singular) points.

The crucial distinction between initial value problems (Chapter 16) and two point boundary value problems (this chapter) is that in the former case we are able to start an acceptable solution at its beginning (initial values) and just march it along by numerical integration to its end (final values); while in the present case, the boundary conditions at the starting point do not determine a unique solution to start with — and a "random" choice among the solutions that satisfy these (incomplete) starting boundary conditions is almost certain *not* to satisfy the boundary conditions at the other specified point(s).

It should not surprise you that iteration is in general required to meld these spatially scattered boundary conditions into a single global solution of the differential equations. For this reason, two point boundary value problems require considerably more effort to solve than do initial value problems. You have to integrate your differential equations over the interval of interest, or perform an analogous "relaxation" procedure (see below), at least several, and sometimes very many, times. Only in the special case of linear differential equations can you say in advance just how many such iterations will be required.

The "standard" two point boundary value problem has the following form: We desire the solution to a set of N coupled first-order ordinary differential equations, satisfying n_1 boundary conditions at the starting point x_1, and a remaining set of $n_2 = N - n_1$ boundary conditions at the final point x_2. (Recall that all differential equations of order higher than first can be written as coupled sets of first-order equations, cf. §16.0.)

The differential equations are

$$\frac{dy_i(x)}{dx} = g_i(x, y_1, y_2, \ldots, y_N) \qquad i = 1, 2, \ldots, N \tag{17.0.1}$$

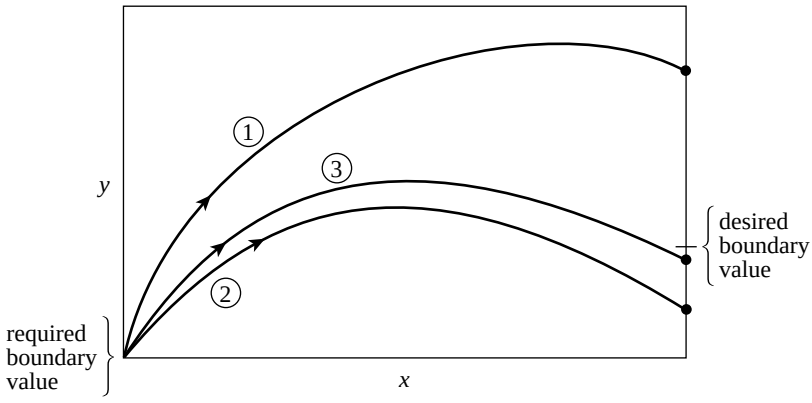


Figure 17.0.1. Shooting method (schematic). Trial integrations that satisfy the boundary condition at one endpoint are "launched." The discrepancies from the desired boundary condition at the other endpoint are used to adjust the starting conditions, until boundary conditions at both endpoints are ultimately satisfied.

At x_1, the solution is supposed to satisfy

$$B_{1j}(x_1, y_1, y_2, \ldots, y_N) = 0 \qquad j = 1, \ldots, n_1 \tag{17.0.2}$$

while at x_2, it is supposed to satisfy

$$B_{2k}(x_2, y_1, y_2, \ldots, y_N) = 0 \qquad k = 1, \ldots, n_2 \tag{17.0.3}$$

There are two distinct classes of numerical methods for solving two point boundary value problems. In the *shooting method* (§17.1) we choose values for all of the dependent variables at one boundary. These values must be consistent with any boundary conditions for *that* boundary, but otherwise are arranged to depend on arbitrary free parameters whose values we initially "randomly" guess. We then integrate the ODEs by initial value methods, arriving at the other boundary (and/or any interior points with boundary conditions specified). In general, we find discrepancies from the desired boundary values there. Now we have a multidimensional root-finding problem, as was treated in §9.6 and §9.7: Find the adjustment of the free parameters at the starting point that zeros the discrepancies at the other boundary point(s). If we liken integrating the differential equations to following the trajectory of a shot from gun to target, then picking the initial conditions corresponds to aiming (see Figure 17.0.1). The shooting method provides a systematic approach to taking a set of "ranging" shots that allow us to improve our "aim" systematically.

As another variant of the shooting method (§17.2), we can guess unknown free parameters at both ends of the domain, integrate the equations to a common midpoint, and seek to adjust the guessed parameters so that the solution joins "smoothly" at the fitting point. In all shooting methods, trial solutions satisfy the differential equations "exactly" (or as exactly as we care to make our numerical integration), but the trial solutions come to satisfy the required boundary conditions only after the iterations are finished.

Relaxation methods use a different approach. The differential equations are replaced by finite-difference equations on a mesh of points that covers the range of

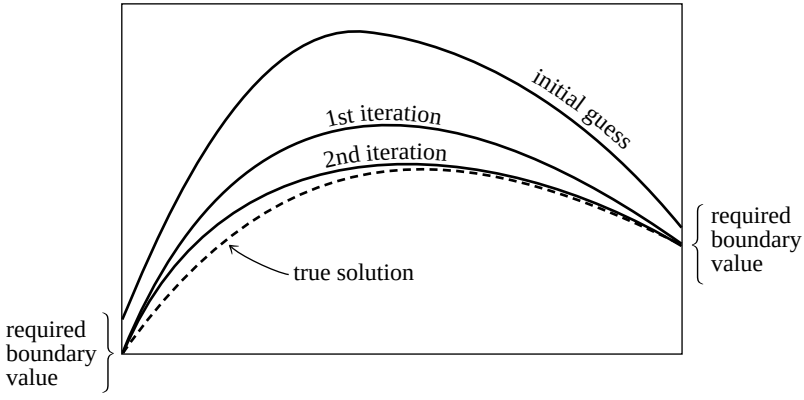


Figure 17.0.2. Relaxation method (schematic). An initial solution is guessed that approximately satisfies the differential equation and boundary conditions. An iterative process adjusts the function to bring it into close agreement with the true solution.

the integration. A trial solution consists of values for the dependent variables at each mesh point, *not* satisfying the desired finite-difference equations, nor necessarily even satisfying the required boundary conditions. The iteration, now called *relaxation*, consists of adjusting all the values on the mesh so as to bring them into successively closer agreement with the finite-difference equations and, simultaneously, with the boundary conditions (see Figure 17.0.2). For example, if the problem involves three coupled equations and a mesh of one hundred points, we must guess and improve three hundred variables representing the solution.

With all this adjustment, you may be surprised that relaxation is ever an efficient method, but (for the right problems) it really is! Relaxation works better than shooting when the boundary conditions are especially delicate or subtle, or where they involve complicated algebraic relations that cannot easily be solved in closed form. Relaxation works best when the solution is smooth and not highly oscillatory. Such oscillations would require many grid points for accurate representation. The number and position of required points may not be known *a priori*. Shooting methods are usually preferred in such cases, because their variable stepsize integrations adjust naturally to a solution's peculiarities.

Relaxation methods are often preferred when the ODEs have extraneous solutions which, while not appearing in the final solution satisfying all boundary conditions, may wreak havoc on the initial value integrations required by shooting. The typical case is that of trying to maintain a dying exponential in the presence of growing exponentials.

Good initial guesses are the secret of efficient relaxation methods. Often one has to solve a problem many times, each time with a slightly different value of some parameter. In that case, the previous solution is usually a good initial guess when the parameter is changed, and relaxation will work well.

Until you have enough experience to make your own judgment between the two methods, you might wish to follow the advice of your authors, who are notorious computer gunslingers: We always shoot first, and only then relax.

Problems Reducible to the Standard Boundary Problem

There are two important problems that can be reduced to the standard boundary value problem described by equations (17.0.1) – (17.0.3). The first is the *eigenvalue problem for differential equations*. Here the right-hand side of the system of differential equations depends on a parameter λ,

$$\frac{dy_i(x)}{dx} = g_i(x, y_1, \ldots, y_N, \lambda) \tag{17.0.4}$$

and one has to satisfy $N + 1$ boundary conditions instead of just N. The problem is overdetermined and in general there is no solution for arbitrary values of λ. For certain special values of λ, the eigenvalues, equation (17.0.4) does have a solution.

We reduce this problem to the standard case by introducing a new dependent variable

$$y_{N+1} \equiv \lambda \tag{17.0.5}$$

and another differential equation

$$\frac{dy_{N+1}}{dx} = 0 \tag{17.0.6}$$

An example of this trick is given in §17.4.

The other case that can be put in the standard form is a *free boundary problem*. Here only one boundary abscissa x_1 is specified, while the other boundary x_2 is to be determined so that the system (17.0.1) has a solution satisfying a total of $N + 1$ boundary conditions. Here we again add an extra constant dependent variable:

$$y_{N+1} \equiv x_2 - x_1 \tag{17.0.7}$$

$$\frac{dy_{N+1}}{dx} = 0 \tag{17.0.8}$$

We also define a new *independent* variable t by setting

$$x - x_1 \equiv t\, y_{N+1}, \qquad 0 \le t \le 1 \tag{17.0.9}$$

The system of $N + 1$ differential equations for dy_i/dt is now in the standard form, with t varying between the known limits 0 and 1.

CITED REFERENCES AND FURTHER READING:

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

Eggleton, P.P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364.

London, R.A., and Flannery, B.P. 1982, *Astrophysical Journal*, vol. 258, pp. 260–269.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§7.3–7.4.

17.1 The Shooting Method

In this section we discuss "pure" shooting, where the integration proceeds from x_1 to x_2, and we try to match boundary conditions at the end of the integration. In the next section, we describe shooting to an intermediate fitting point, where the solution to the equations and boundary conditions is found by launching "shots" from both sides of the interval and trying to match continuity conditions at some intermediate point.

Our implementation of the shooting method exactly implements multidimensional, globally convergent Newton-Raphson (§9.7). It seeks to zero n_2 functions of n_2 variables. The functions are obtained by integrating N differential equations from x_1 to x_2. Let us see how this works:

At the starting point x_1 there are N starting values y_i to be specified, but subject to n_1 conditions. Therefore there are $n_2 = N - n_1$ *freely specifiable* starting values. Let us imagine that these freely specifiable values are the components of a vector $\mathbf{V}$ that lives in a vector space of dimension n_2. Then you, the user, knowing the functional form of the boundary conditions (17.0.2), can write a subroutine that generates a complete set of N starting values $\mathbf{y}$, satisfying the boundary conditions at x_1, from an arbitrary vector value of $\mathbf{V}$ in which there are no restrictions on the n_2 component values. In other words, (17.0.2) converts to a prescription

$$y_i(x_1) = y_i(x_1; V_1, \ldots, V_{n_2}) \qquad i = 1, \ldots, N \tag{17.1.1}$$

Below, the subroutine that implements (17.1.1) will be called `load`.

Notice that the components of $\mathbf{V}$ might be exactly the values of certain "free" components of $\mathbf{y}$, with the other components of $\mathbf{y}$ determined by the boundary conditions. Alternatively, the components of $\mathbf{V}$ might parametrize the solutions that satisfy the starting boundary conditions in some other convenient way. Boundary conditions often impose algebraic relations among the y_i, rather than specific values for each of them. Using some auxiliary set of parameters often makes it easier to "solve" the boundary relations for a consistent set of y_i's. It makes no difference which way you go, as long as your vector space of $\mathbf{V}$'s generates (through 17.1.1) all allowed starting vectors $\mathbf{y}$.

Given a particular $\mathbf{V}$, a particular $\mathbf{y}(x_1)$ is thus generated. It can then be turned into a $\mathbf{y}(x_2)$ by integrating the ODEs to x_2 as an initial value problem (e.g., using Chapter 16's `odeint`). Now, at x_2, let us define a *discrepancy vector* $\mathbf{F}$, also of dimension n_2, whose components measure how far we are from satisfying the n_2 boundary conditions at x_2 (17.0.3). Simplest of all is just to use the right-hand sides of (17.0.3),

$$F_k = B_{2k}(x_2, \mathbf{y}) \qquad k = 1, \ldots, n_2 \tag{17.1.2}$$

As in the case of $\mathbf{V}$, however, you can use any other convenient parametrization, as long as your space of $\mathbf{F}$'s spans the space of possible discrepancies from the desired boundary conditions, with all components of $\mathbf{F}$ equal to zero if and only if the boundary conditions at x_2 are satisfied. Below, you will be asked to supply a user-written subroutine `score` which uses (17.0.3) to convert an N-vector of ending values $\mathbf{y}(x_2)$ into an n_2-vector of discrepancies $\mathbf{F}$.

17.1 The Shooting Method

In this section we discuss "pure" shooting, where the integration proceeds from x_1 to x_2, and we try to match boundary conditions at the end of the integration. In the next section, we describe shooting to an intermediate fitting point, where the solution to the equations and boundary conditions is found by launching "shots" from both sides of the interval and trying to match continuity conditions at some intermediate point.

Our implementation of the shooting method exactly implements multidimensional, globally convergent Newton-Raphson (§9.7). It seeks to zero n_2 functions of n_2 variables. The functions are obtained by integrating N differential equations from x_1 to x_2. Let us see how this works:

At the starting point x_1 there are N starting values y_i to be specified, but subject to n_1 conditions. Therefore there are $n_2 = N - n_1$ *freely specifiable* starting values. Let us imagine that these freely specifiable values are the components of a vector $\mathbf{V}$ that lives in a vector space of dimension n_2. Then you, the user, knowing the functional form of the boundary conditions (17.0.2), can write a subroutine that generates a complete set of N starting values $\mathbf{y}$, satisfying the boundary conditions at x_1, from an arbitrary vector value of $\mathbf{V}$ in which there are no restrictions on the n_2 component values. In other words, (17.0.2) converts to a prescription

$$y_i(x_1) = y_i(x_1; V_1, \ldots, V_{n_2}) \qquad i = 1, \ldots, N \tag{17.1.1}$$

Below, the subroutine that implements (17.1.1) will be called `load`.

Notice that the components of $\mathbf{V}$ might be exactly the values of certain "free" components of $\mathbf{y}$, with the other components of $\mathbf{y}$ determined by the boundary conditions. Alternatively, the components of $\mathbf{V}$ might parametrize the solutions that satisfy the starting boundary conditions in some other convenient way. Boundary conditions often impose algebraic relations among the y_i, rather than specific values for each of them. Using some auxiliary set of parameters often makes it easier to "solve" the boundary relations for a consistent set of y_i's. It makes no difference which way you go, as long as your vector space of $\mathbf{V}$'s generates (through 17.1.1) all allowed starting vectors $\mathbf{y}$.

Given a particular $\mathbf{V}$, a particular $\mathbf{y}(x_1)$ is thus generated. It can then be turned into a $\mathbf{y}(x_2)$ by integrating the ODEs to x_2 as an initial value problem (e.g., using Chapter 16's `odeint`). Now, at x_2, let us define a *discrepancy vector* $\mathbf{F}$, also of dimension n_2, whose components measure how far we are from satisfying the n_2 boundary conditions at x_2 (17.0.3). Simplest of all is just to use the right-hand sides of (17.0.3),

$$F_k = B_{2k}(x_2, \mathbf{y}) \qquad k = 1, \ldots, n_2 \tag{17.1.2}$$

As in the case of $\mathbf{V}$, however, you can use any other convenient parametrization, as long as your space of $\mathbf{F}$'s spans the space of possible discrepancies from the desired boundary conditions, with all components of $\mathbf{F}$ equal to zero if and only if the boundary conditions at x_2 are satisfied. Below, you will be asked to supply a user-written subroutine `score` which uses (17.0.3) to convert an N-vector of ending values $\mathbf{y}(x_2)$ into an n_2-vector of discrepancies $\mathbf{F}$.

Now, as far as Newton-Raphson is concerned, we are nearly in business. We want to find a vector value of $\mathbf{V}$ that zeros the vector value of $\mathbf{F}$. We do this by invoking the globally convergent Newton's method implemented in the routine `newt` of §9.7. Recall that the heart of Newton's method involves solving the set of n_2 linear equations

$$\mathbf{J} \cdot \delta\mathbf{V} = -\mathbf{F} \tag{17.1.3}$$

and then adding the correction back,

$$\mathbf{V}^{\text{new}} = \mathbf{V}^{\text{old}} + \delta\mathbf{V} \tag{17.1.4}$$

In (17.1.3), the Jacobian matrix $\mathbf{J}$ has components given by

$$J_{ij} = \frac{\partial F_i}{\partial V_j} \tag{17.1.5}$$

It is not feasible to compute these partial derivatives analytically. Rather, each requires a *separate* integration of the N ODEs, followed by the evaluation of

$$\frac{\partial F_i}{\partial V_j} \approx \frac{F_i(V_1, \ldots, V_j + \Delta V_j, \ldots) - F_i(V_1, \ldots, V_j, \ldots)}{\Delta V_j} \tag{17.1.6}$$

This is done automatically for you in the routine `fdjac` that comes with `newt`. The only input to `newt` that you have to provide is the routine `funcv` that calculates $\mathbf{F}$ by integrating the ODEs. Here is the appropriate routine:

```
C   SUBROUTINE shoot(n2,v,f) is named "funcv" for use with "newt"
    SUBROUTINE funcv(n2,v,f)
    INTEGER n2,nvar,kmax,kount,KMAXX,NMAX
    REAL f(n2),v(n2),x1,x2,dxsav,xp,yp,EPS
    PARAMETER (NMAX=50,KMAXX=200,EPS=1.e-6)    At most NMAX coupled ODEs.
    COMMON /caller/ x1,x2,nvar
    COMMON /path/ kmax,kount,dxsav,xp(KMAXX),yp(NMAX,KMAXX)
C   USES derivs,load,odeint,rkqs,score
        Routine for use with newt to solve a two point boundary value problem for nvar coupled
        ODEs by shooting from x1 to x2. Initial values for the nvar ODEs at x1 are generated
        from the n2 input coefficients v(1:n2), using the user-supplied routine load. The routine
        integrates the ODEs to x2 using the Runge-Kutta method with tolerance EPS, initial stepsize
        h1, and minimum stepsize hmin. At x2 it calls the user-supplied subroutine score to
        evaluate the n2 functions f(1:n2) that ought to be zero to satisfy the boundary conditions
        at x2. The functions f are returned on output. newt uses a globally convergent Newton's
        method to adjust the values of v until the functions f are zero. The user-supplied subroutine
        derivs(x,y,dydx) supplies derivative information to the ODE integrator (see Chapter
        16). The common block caller receives its values from the main program so that funcv
        can have the syntax required by newt. The common block path is included for compatibility
        with odeint.
    INTEGER nbad,nok
    REAL h1,hmin,y(NMAX)
    EXTERNAL derivs,rkqs
    kmax=0
    h1=(x2-x1)/100.
    hmin=0.
    call load(x1,v,y)
    call odeint(y,nvar,x1,x2,EPS,h1,hmin,nok,nbad,derivs,rkqs)
    call score(x2,y,f)
    return
    END
```

For some problems the initial stepsize ΔV might depend sensitively upon the initial conditions. It is straightforward to alter `load` to include a suggested stepsize `h1` as another returned argument and feed it to `fdjac` via a common block.

A complete cycle of the shooting method thus requires $n_2 + 1$ integrations of the N coupled ODEs: one integration to evaluate the current degree of mismatch, and n_2 for the partial derivatives. Each new cycle requires a new round of $n_2 + 1$ integrations. This illustrates the enormous extra effort involved in solving two point boundary value problems compared with intial value problems.

If the differential equations are *linear*, then only one complete cycle is required, since (17.1.3)–(17.1.4) should take us right to the solution. A second round can be useful, however, in mopping up some (never all) of the roundoff error.

As given here, `shoot` uses the quality controlled Runge-Kutta method of §16.2 to integrate the ODEs, but any of the other methods of Chapter 16 could just as well be used.

You, the user, must supply `shoot` with: (i) a subroutine `load(x1,v,y)` which returns the `n`-vector `y(1:n)` (satisfying the starting boundary conditions, of course), given the freely specifiable variables of `v(1:n2)` at the initial point `x1`; (ii) a subroutine `score(x2,y,f)` which returns the discrepancy vector `f(1:n2)` of the ending boundary conditions, given the vector `y(1:n)` at the endpoint `x2`; (iii) a starting vector `v(1:n2)`; (iv) a subroutine `derivs` for the ODE integration; and other obvious parameters as described in the header comment above.

In §17.4 we give a sample program illustrating how to use `shoot`.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

17.2 Shooting to a Fitting Point

The shooting method described in §17.1 tacitly assumed that the "shots" would be able to traverse the entire domain of integration, even at the early stages of convergence to a correct solution. In some problems it can happen that, for very wrong starting conditions, an initial solution can't even get from x_1 to x_2 without encountering some incalculable, or catastrophic, result. For example, the argument of a square root might go negative, causing the numerical code to crash. Simple shooting would be stymied.

A different, but related, case is where the endpoints are both singular points of the set of ODEs. One frequently needs to use special methods to integrate near the singular points, analytic asymptotic expansions, for example. In such cases it is feasible to integrate in the direction *away* from a singular point, using the special method to get through the first little bit and then reading off "initial" values for further numerical integration. However it is usually not feasible to integrate *into* a singular point, if only because one has not usually expended the same analytic

For some problems the initial stepsize ΔV might depend sensitively upon the initial conditions. It is straightforward to alter `load` to include a suggested stepsize `h1` as another returned argument and feed it to `fdjac` via a common block.

A complete cycle of the shooting method thus requires $n_2 + 1$ integrations of the N coupled ODEs: one integration to evaluate the current degree of mismatch, and n_2 for the partial derivatives. Each new cycle requires a new round of $n_2 + 1$ integrations. This illustrates the enormous extra effort involved in solving two point boundary value problems compared with intial value problems.

If the differential equations are *linear*, then only one complete cycle is required, since (17.1.3)–(17.1.4) should take us right to the solution. A second round can be useful, however, in mopping up some (never all) of the roundoff error.

As given here, `shoot` uses the quality controlled Runge-Kutta method of §16.2 to integrate the ODEs, but any of the other methods of Chapter 16 could just as well be used.

You, the user, must supply `shoot` with: (i) a subroutine `load(x1,v,y)` which returns the `n`-vector `y(1:n)` (satisfying the starting boundary conditions, of course), given the freely specifiable variables of `v(1:n2)` at the initial point `x1`; (ii) a subroutine `score(x2,y,f)` which returns the discrepancy vector `f(1:n2)` of the ending boundary conditions, given the vector `y(1:n)` at the endpoint `x2`; (iii) a starting vector `v(1:n2)`; (iv) a subroutine `derivs` for the ODE integration; and other obvious parameters as described in the header comment above.

In §17.4 we give a sample program illustrating how to use `shoot`.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

17.2 Shooting to a Fitting Point

The shooting method described in §17.1 tacitly assumed that the "shots" would be able to traverse the entire domain of integration, even at the early stages of convergence to a correct solution. In some problems it can happen that, for very wrong starting conditions, an initial solution can't even get from x_1 to x_2 without encountering some incalculable, or catastrophic, result. For example, the argument of a square root might go negative, causing the numerical code to crash. Simple shooting would be stymied.

A different, but related, case is where the endpoints are both singular points of the set of ODEs. One frequently needs to use special methods to integrate near the singular points, analytic asymptotic expansions, for example. In such cases it is feasible to integrate in the direction *away* from a singular point, using the special method to get through the first little bit and then reading off "initial" values for further numerical integration. However it is usually not feasible to integrate *into* a singular point, if only because one has not usually expended the same analytic

effort to obtain expansions of "wrong" solutions near the singular point (those not satisfying the desired boundary condition).

The solution to the above mentioned difficulties is *shooting to a fitting point*. Instead of integrating from x_1 to x_2, we integrate first from x_1 to some point x_f that is *between* x_1 and x_2; and second from x_2 (in the opposite direction) to x_f.

If (as before) the number of boundary conditions imposed at x_1 is n_1, and the number imposed at x_2 is n_2, then there are n_2 freely specifiable starting values at x_1 and n_1 freely specifiable starting values at x_2. (If you are confused by this, go back to §17.1.) We can therefore define an n_2-vector $\mathbf{V}_{(1)}$ of starting parameters at x_1, and a prescription `load1(x1,v1,y)` for mapping $\mathbf{V}_{(1)}$ into a $\mathbf{y}$ that satisfies the boundary conditions at x_1,

$$y_i(x_1) = y_i(x_1; V_{(1)1}, \ldots, V_{(1)n_2}) \qquad i = 1, \ldots, N \tag{17.2.1}$$

Likewise we can define an n_1-vector $\mathbf{V}_{(2)}$ of starting parameters at x_2, and a prescription `load2(x2,v2,y)` for mapping $\mathbf{V}_{(2)}$ into a $\mathbf{y}$ that satisfies the boundary conditions at x_2,

$$y_i(x_2) = y_i(x_2; V_{(2)1}, \ldots, V_{(2)n_1}) \qquad i = 1, \ldots, N \tag{17.2.2}$$

We thus have a total of N freely adjustable parameters in the combination of $\mathbf{V}_{(1)}$ and $\mathbf{V}_{(2)}$. The N conditions that must be satisfied are that there be agreement in N components of $\mathbf{y}$ at x_f between the values obtained integrating from one side and from the other,

$$y_i(x_f; \mathbf{V}_{(1)}) = y_i(x_f; \mathbf{V}_{(2)}) \qquad i = 1, \ldots, N \tag{17.2.3}$$

In some problems, the N matching conditions can be better described (physically, mathematically, or numerically) by using N different functions $F_i,\ i = 1 \ldots N$, each possibly depending on the N components y_i. In those cases, (17.2.3) is replaced by

$$F_i[\mathbf{y}(x_f; \mathbf{V}_{(1)})] = F_i[\mathbf{y}(x_f; \mathbf{V}_{(2)})] \qquad i = 1, \ldots, N \tag{17.2.4}$$

In the program below, the user-supplied subroutine `score(xf,y,f)` is supposed to map an input N-vector $\mathbf{y}$ into an output N-vector $\mathbf{F}$. In most cases, you can dummy this subroutine as the identity mapping.

Shooting to a fitting point uses globally convergent Newton-Raphson exactly as in §17.1. Comparing closely with the routine `shoot` of the previous section, you should have no difficulty in understanding the following routine `shootf`. The main differences in use are that you have to supply both `load1` and `load2`. Also, in the calling program you must supply initial guesses for `v1(1:n2)` and `v2(1:n1)`. Once again a sample program illustrating shooting to a fitting point is given in §17.4.

```
C   SUBROUTINE shootf(n,v,f) is named "funcv" for use with "newt"
    SUBROUTINE funcv(n,v,f)
    INTEGER n,nvar,nn2,kmax,kount,KMAXX,NMAX
    REAL f(n),v(n),x1,x2,xf,dxsav,xp,yp,EPS
    PARAMETER (NMAX=50,KMAXX=200,EPS=1.e-6)   At most NMAX equations.
    COMMON /caller/ x1,x2,xf,nvar,nn2
    COMMON /path/ kmax,kount,dxsav,xp(KMAXX),yp(NMAX,KMAXX)
C   USES derivs,load1,load2,odeint,rkqs,score
```

Routine for use with `newt` to solve a two point boundary value problem for `nvar` coupled ODEs by shooting from `x1` and `x2` to a fitting point `xf`. Initial values for the `nvar`

ODEs at x1 (x2) are generated from the n2 (n1) coefficients v1 (v2), using the user-supplied routine load1 (load2). The coefficients v1 and v2 should be stored in a single array v(1:n1+n2) in the main program by an EQUIVALENCE statement of the form (v1(1),v(1)),(v2(1),v(n2+1)). The input parameter n = n1+n2 = nvar. The routine integrates the ODEs to xf using the Runge-Kutta method with tolerance EPS, initial stepsize h1, and minimum stepsize hmin. At xf it calls the user-supplied subroutine score to evaluate the nvar functions f1 and f2 that ought to match at xf. The differences f are returned on output. newt uses a globally convergent Newton's method to adjust the values of v until the functions f are zero. The user-supplied subroutine derivs(x,y,dydx) supplies derivative information to the ODE integrator (see Chapter 16). The common block caller receives its values from the main program so that funcv can have the syntax required by newt. Set nn2 = n2 in the main program. The common block path is for compatibility with odeint.

```
INTEGER i,nbad,nok
REAL h1,hmin,f1(NMAX),f2(NMAX),y(NMAX)
EXTERNAL derivs,rkqs
kmax=0
h1=(x2-x1)/100.
hmin=0.
call load1(x1,v,y)                    Path from x1 to xf with best trial values v1.
call odeint(y,nvar,x1,xf,EPS,h1,hmin,nok,nbad,derivs,rkqs)
call score(xf,y,f1)
call load2(x2,v(nn2+1),y)             Path from x2 to xf with best trial values v2.
call odeint(y,nvar,x2,xf,EPS,h1,hmin,nok,nbad,derivs,rkqs)
call score(xf,y,f2)
do 11 i=1,n
    f(i)=f1(i)-f2(i)
enddo 11
return
END
```

There are boundary value problems where even shooting to a fitting point fails — the integration interval has to be partitioned by several fitting points with the solution being matched at each such point. For more details see [1].

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§7.3.5–7.3.6. [1]

17.3 Relaxation Methods

In *relaxation methods* we replace ODEs by approximate *finite-difference equations* (FDEs) on a grid or mesh of points that spans the domain of interest. As a typical example, we could replace a general first-order differential equation

$$\frac{dy}{dx} = g(x, y) \tag{17.3.1}$$

with an algebraic equation relating function values at two points $k, k-1$:

$$y_k - y_{k-1} - (x_k - x_{k-1})\, g\left[\tfrac{1}{2}(x_k + x_{k-1}), \tfrac{1}{2}(y_k + y_{k-1})\right] = 0 \tag{17.3.2}$$

ODEs at x1 (x2) are generated from the n2 (n1) coefficients v1 (v2), using the user-supplied routine load1 (load2). The coefficients v1 and v2 should be stored in a single array v(1:n1+n2) in the main program by an EQUIVALENCE statement of the form (v1(1),v(1)),(v2(1),v(n2+1)). The input parameter n = n1+n2 = nvar. The routine integrates the ODEs to xf using the Runge-Kutta method with tolerance EPS, initial stepsize h1, and minimum stepsize hmin. At xf it calls the user-supplied subroutine score to evaluate the nvar functions f1 and f2 that ought to match at xf. The differences f are returned on output. newt uses a globally convergent Newton's method to adjust the values of v until the functions f are zero. The user-supplied subroutine derivs(x,y,dydx) supplies derivative information to the ODE integrator (see Chapter 16). The common block caller receives its values from the main program so that funcv can have the syntax required by newt. Set nn2 = n2 in the main program. The common block path is for compatibility with odeint.

```
INTEGER i,nbad,nok
REAL h1,hmin,f1(NMAX),f2(NMAX),y(NMAX)
EXTERNAL derivs,rkqs
kmax=0
h1=(x2-x1)/100.
hmin=0.
call load1(x1,v,y)                    Path from x1 to xf with best trial values v1.
call odeint(y,nvar,x1,xf,EPS,h1,hmin,nok,nbad,derivs,rkqs)
call score(xf,y,f1)
call load2(x2,v(nn2+1),y)             Path from x2 to xf with best trial values v2.
call odeint(y,nvar,x2,xf,EPS,h1,hmin,nok,nbad,derivs,rkqs)
call score(xf,y,f2)
do 11 i=1,n
    f(i)=f1(i)-f2(i)
enddo 11
return
END
```

There are boundary value problems where even shooting to a fitting point fails — the integration interval has to be partitioned by several fitting points with the solution being matched at each such point. For more details see [1].

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§7.3.5–7.3.6. [1]

17.3 Relaxation Methods

In *relaxation methods* we replace ODEs by approximate *finite-difference equations* (FDEs) on a grid or mesh of points that spans the domain of interest. As a typical example, we could replace a general first-order differential equation

$$\frac{dy}{dx} = g(x, y) \tag{17.3.1}$$

with an algebraic equation relating function values at two points $k, k-1$:

$$y_k - y_{k-1} - (x_k - x_{k-1})\, g\left[\tfrac{1}{2}(x_k + x_{k-1}), \tfrac{1}{2}(y_k + y_{k-1})\right] = 0 \tag{17.3.2}$$

The form of the FDE in (17.3.2) illustrates the idea, but not uniquely: There are many ways to turn the ODE into an FDE. When the problem involves N coupled first-order ODEs represented by FDEs on a mesh of M points, a solution consists of values for N dependent functions given at each of the M mesh points, or $N \times M$ variables in all. The relaxation method determines the solution by starting with a guess and improving it, iteratively. As the iterations improve the solution, the result is said to *relax* to the true solution.

While several iteration schemes are possible, for most problems our old standby, multi-dimensional Newton's method, works well. The method produces a matrix equation that must be solved, but the matrix takes a special, "block diagonal" form, that allows it to be inverted far more economically both in time and storage than would be possible for a general matrix of size $(MN) \times (MN)$. Since MN can easily be several thousand, this is crucial for the feasibility of the method.

Our implementation couples at most pairs of points, as in equation (17.3.2). More points can be coupled, but then the method becomes more complex. We will provide enough background so that you can write a more general scheme if you have the patience to do so.

Let us develop a general set of algebraic equations that represent the ODEs by FDEs. The ODE problem is exactly identical to that expressed in equations (17.0.1)–(17.0.3) where we had N coupled first-order equations that satisfy n_1 boundary conditions at x_1 and $n_2 = N - n_1$ boundary conditions at x_2. We first define a mesh or grid by a set of $k = 1, 2, ..., M$ points at which we supply values for the independent variable x_k. In particular, x_1 is the initial boundary, and x_M is the final boundary. We use the notation $\mathbf{y}_k$ to refer to the entire set of dependent variables $y_1, y_2, \ldots, y_N$ at point x_k. At an arbitrary point k in the middle of the mesh, we approximate the set of N first-order ODEs by algebraic relations of the form

$$0 = \mathbf{E}_k \equiv \mathbf{y}_k - \mathbf{y}_{k-1} - (x_k - x_{k-1})\mathbf{g}_k(x_k, x_{k-1}, \mathbf{y}_k, \mathbf{y}_{k-1}), \quad k = 2, 3, \ldots, M \tag{17.3.3}$$

The notation signifies that $\mathbf{g}_k$ can be evaluated using information from both points $k, k-1$. The FDEs labeled by $\mathbf{E}_k$ provide N equations coupling $2N$ variables at points $k, k-1$. There are $M-1$ points, $k = 2, 3, \ldots, M$, at which difference equations of the form (17.3.3) apply. Thus the FDEs provide a total of $(M-1)N$ equations for the MN unknowns. The remaining N equations come from the boundary conditions.

At the first boundary we have

$$0 = \mathbf{E}_1 \equiv \mathbf{B}(x_1, \mathbf{y}_1) \tag{17.3.4}$$

while at the second boundary

$$0 = \mathbf{E}_{M+1} \equiv \mathbf{C}(x_M, \mathbf{y}_M) \tag{17.3.5}$$

The vectors $\mathbf{E}_1$ and $\mathbf{B}$ have only n_1 nonzero components, corresponding to the n_1 boundary conditions at x_1. It will turn out to be useful to take these nonzero components to be the *last* n_1 components. In other words, $E_{j,1} \neq 0$ only for $j = n_2 + 1, n_2 + 2, \ldots, N$. At the other boundary, only the first n_2 components of $\mathbf{E}_{M+1}$ and $\mathbf{C}$ are nonzero: $E_{j,M+1} \neq 0$ only for $j = 1, 2, \ldots, n_2$.

The "solution" of the FDE problem in (17.3.3)–(17.3.5) consists of a set of variables $y_{j,k}$, the values of the N variables y_j at the M points x_k. The algorithm we describe below requires an initial guess for the $y_{j,k}$. We then determine increments $\Delta y_{j,k}$ such that $y_{j,k} + \Delta y_{j,k}$ is an improved approximation to the solution.

Equations for the increments are developed by expanding the FDEs in first-order Taylor series with respect to small changes $\Delta \mathbf{y}_k$. At an interior point, $k = 2, 3, \ldots, M$ this gives:

$$\begin{aligned} \mathbf{E}_k(\mathbf{y}_k + \Delta\mathbf{y}_k, \mathbf{y}_{k-1} + \Delta\mathbf{y}_{k-1}) &\approx \mathbf{E}_k(\mathbf{y}_k, \mathbf{y}_{k-1}) \\ &+ \sum_{n=1}^{N} \frac{\partial \mathbf{E}_k}{\partial y_{n,k-1}} \Delta y_{n,k-1} + \sum_{n=1}^{N} \frac{\partial \mathbf{E}_k}{\partial y_{n,k}} \Delta y_{n,k} \end{aligned} \tag{17.3.6}$$

For a solution we want the updated value $\mathbf{E}(\mathbf{y} + \Delta\mathbf{y})$ to be zero, so the general set of equations at an interior point can be written in matrix form as

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,k-1} + \sum_{n=N+1}^{2N} S_{j,n} \Delta y_{n-N,k} = -E_{j,k}, \quad j = 1, 2, \ldots, N \tag{17.3.7}$$

where

$$S_{j,n} = \frac{\partial E_{j,k}}{\partial y_{n,k-1}}, \quad S_{j,n+N} = \frac{\partial E_{j,k}}{\partial y_{n,k}}, \quad n = 1, 2, \ldots, N \tag{17.3.8}$$

The quantity $S_{j,n}$ is an $N \times 2N$ matrix at each point k. Each interior point thus supplies a block of N equations coupling $2N$ corrections to the solution variables at the points $k, k-1$.

Similarly, the algebraic relations at the boundaries can be expanded in a first-order Taylor series for increments that improve the solution. Since $\mathbf{E}_1$ depends only on $\mathbf{y}_1$, we find at the first boundary:

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,1} = -E_{j,1}, \quad j = n_2 + 1, n_2 + 2, \ldots, N \tag{17.3.9}$$

where

$$S_{j,n} = \frac{\partial E_{j,1}}{\partial y_{n,1}}, \quad n = 1, 2, \ldots, N \tag{17.3.10}$$

At the second boundary,

$$\sum_{n=1}^{N} S_{j,n} \Delta y_{n,M} = -E_{j,M+1}, \quad j = 1, 2, \ldots, n_2 \tag{17.3.11}$$

where

$$S_{j,n} = \frac{\partial E_{j,M+1}}{\partial y_{n,M}}, \quad n = 1, 2, \ldots, N \tag{17.3.12}$$

We thus have in equations (17.3.7)–(17.3.12) a set of linear equations to be solved for the corrections $\Delta\mathbf{y}$, iterating until the corrections are sufficiently small. The equations have a special structure, because each $S_{j,n}$ couples only points $k, k-1$. Figure 17.3.1 illustrates the typical structure of the complete matrix equation for the case of 5 variables and 4 mesh points, with 3 boundary conditions at the first boundary and 2 at the second. The 3×5 block of nonzero entries in the top left-hand corner of the matrix comes from the boundary condition $S_{j,n}$ at point $k = 1$. The next three 5×10 blocks are the $S_{j,n}$ at the interior points, coupling variables at mesh points (2,1), (3,2), and (4,3). Finally we have the block corresponding to the second boundary condition.

We can solve equations (17.3.7)–(17.3.12) for the increments $\Delta\mathbf{y}$ using a form of Gaussian elimination that exploits the special structure of the matrix to minimize the total number of operations, and that minimizes storage of matrix coefficients by packing the elements in a special blocked structure. (You might wish to review Chapter 2, especially §2.2, if you are unfamiliar with the steps involved in Gaussian elimination.) Recall that Gaussian elimination consists of manipulating the equations by elementary operations such as dividing rows of coefficients by a common factor to produce unity in diagonal elements, and adding appropriate multiples of other rows to produce zeros below the diagonal. Here we take advantage of the block structure by performing a bit more reduction than in pure Gaussian elimination, so that the storage of coefficients is minimized. Figure 17.3.2 shows the form that we wish to achieve by elimination, just prior to the backsubstitution step. Only a small subset of the reduced $MN \times MN$ matrix elements needs to be stored as the elimination progresses. Once the matrix elements reach the stage in Figure 17.3.2, the solution follows quickly by a backsubstitution procedure.

Furthermore, the entire procedure, except the backsubstitution step, operates only on one block of the matrix at a time. The procedure contains four types of operations: (1) partial reduction to zero of certain elements of a block using results from a previous step, (2) elimination of the square structure of the remaining block elements such that the square section contains unity along the diagonal, and zero in off-diagonal elements, (3) storage of the remaining nonzero coefficients for use in later steps, and (4) backsubstitution. We illustrate the steps schematically by figures.

Consider the block of equations describing corrections available from the initial boundary conditions. We have n_1 equations for N unknown corrections. We wish to transform the first

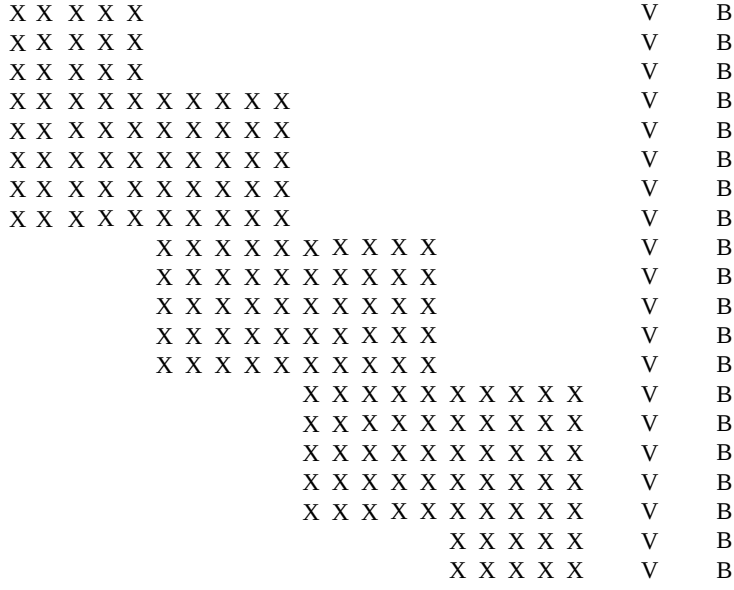


Figure 17.3.1. Matrix structure of a set of linear finite-difference equations (FDEs) with boundary conditions imposed at both endpoints. Here X represents a coefficient of the FDEs, V represents a component of the unknown solution vector, and B is a component of the known right-hand side. Empty spaces represent zeros. The matrix equation is to be solved by a special form of Gaussian elimination. (See text for details.)

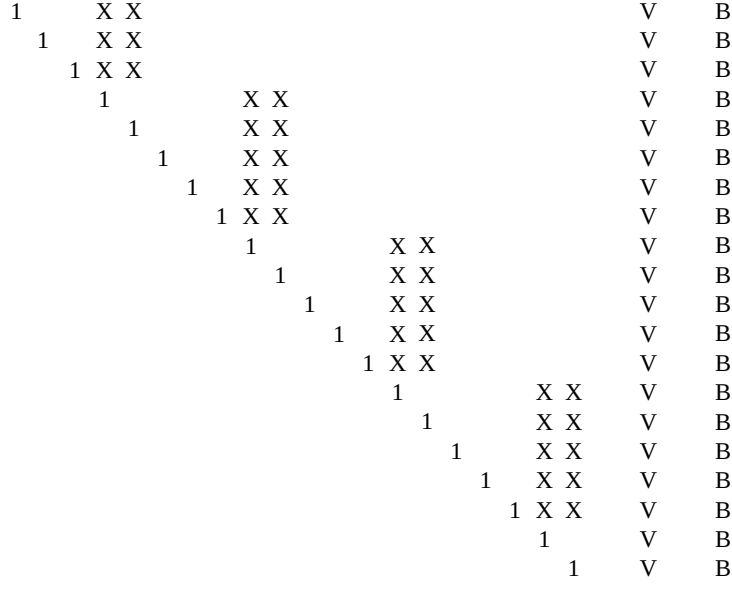


Figure 17.3.2. Target structure of the Gaussian elimination. Once the matrix of Figure 17.3.1 has been reduced to this form, the solution follows quickly by backsubstitution.

block so that its left-hand $n_1 \times n_1$ square section becomes unity along the diagonal, and zero in off-diagonal elements. Figure 17.3.3 shows the original and final form of the first block of the matrix. In the figure we designate matrix elements that are subject to diagonalization by "D", and elements that will be altered by "A"; in the final block, elements that are stored are labeled by "S". We get from start to finish by selecting in turn n_1 "pivot" elements from among the first n_1 columns, normalizing the pivot row so that the value of the "pivot" element is unity, and adding appropriate multiples of this row to the remaining rows so that they contain zeros in the pivot column. In its final form, the reduced block expresses values for the corrections to the first n_1 variables at mesh point 1 in terms of values for the remaining n_2 unknown corrections at point 1, i.e., we now know what the first n_1 elements are in terms of the remaining n_2 elements. We store only the final set of n_2 nonzero columns from the initial block, plus the column for the altered right-hand side of the matrix equation.

We must emphasize here an important detail of the method. To exploit the reduced storage allowed by operating on blocks, it is essential that the ordering of columns in the `s` matrix of derivatives be such that pivot elements can be found among the first n_1 rows of the matrix. This means that the n_1 boundary conditions at the first point must contain some dependence on the first `j=1,2,...,`n_1 dependent variables, `y(j,1)`. If not, then the original square $n_1 \times n_1$ subsection of the first block will appear to be singular, and the method will fail. Alternatively, we would have to allow the search for pivot elements to involve all N columns of the block, and this would require column swapping and far more bookkeeping. The code provides a simple method of reordering the variables, i.e., the columns of the `s` matrix, so that this can be done easily. End of important detail.

Next consider the block of N equations representing the FDEs that describe the relation between the $2N$ corrections at points 2 and 1. The elements of that block, together with results from the previous step, are illustrated in Figure 17.3.4. Note that by adding suitable multiples of rows from the first block we can reduce to zero the first n_1 columns of the block (labeled by "Z"), and, to do so, we will need to alter only the columns from $n_1 + 1$ to N and the vector element on the right-hand side. Of the remaining columns we can diagonalize a square subsection of $N \times N$ elements, labeled by "D" in the figure. In the process we alter the final set of $n_2 + 1$ columns, denoted "A" in the figure. The second half of the figure shows the block when we finish operating on it, with the stored $(n_2 + 1) \times N$ elements labeled by "S."

If we operate on the next set of equations corresponding to the FDEs coupling corrections at points 3 and 2, we see that the state of available results and new equations exactly reproduces the situation described in the previous paragraph. Thus, we can carry out those steps again for each block in turn through block M. Finally on block $M + 1$ we encounter the remaining boundary conditions.

Figure 17.3.5 shows the final block of n_2 FDEs relating the N corrections for variables at mesh point M, together with the result of reducing the previous block. Again, we can first use the prior results to zero the first n_1 columns of the block. Now, when we diagonalize the remaining square section, we strike gold: We get values for the final n_2 corrections at mesh point M.

With the final block reduced, the matrix has the desired form shown previously in Figure 17.3.2, and the matrix is ripe for backsubstitution. Starting with the bottom row and working up towards the top, at each stage we can simply determine one unknown correction in terms of known quantities.

The subroutine `solvde` organizes the steps described above. The principal procedures used in the algorithm are performed by subroutines called internally by `solvde`. The subroutine `red` eliminates leading columns of the `s` matrix using results from prior blocks. `pinvs` diagonalizes the square subsection of `s` and stores unreduced coefficients. `bksub` carries out the backsubstitution step. The user of `solvde` must understand the calling arguments, as described below, and supply a subroutine `difeq`, called by `solvde`, that evaluates the `s` matrix for each block.

Most of the arguments in the call to `solvde` have already been described, but some require discussion. Array `y(j,k)` contains the initial guess for the solution, with `j` labeling the dependent variables at mesh points `k`. The problem involves `ne` FDEs spanning points `k=1,..., m`. `nb` boundary conditions apply at the first point `k=1`. The array `indexv(j)` establishes the correspondence between columns of the `s` matrix, equations (17.3.8), (17.3.10),

```
(a)  D D D A A        V   A
     D D D A A        V   A
     D D D A A        V   A

(b)  1 0 0 S S        V   S
     0 1 0 S S        V   S
     0 0 1 S S        V   S
```

Figure 17.3.3. Reduction process for the first (upper left) block of the matrix in Figure 17.3.1. (a) Original form of the block, (b) final form. (See text for explanation.)

```
(a)  1 0 0 S S                    V   S
     0 1 0 S S                    V   S
     0 0 1 S S                    V   S
     Z Z Z D D D D D A A          V   A
     Z Z Z D D D D D A A          V   A
     Z Z Z D D D D D A A          V   A
     Z Z Z D D D D D A A          V   A
     Z Z Z D D D D D A A          V   A

(b)  1 0 0 S S                    V   S
     0 1 0 S S                    V   S
     0 0 1 S S                    V   S
     0 0 0 1 0 0 0 0 S S          V   S
     0 0 0 0 1 0 0 0 S S          V   S
     0 0 0 0 0 1 0 0 S S          V   S
     0 0 0 0 0 0 1 0 S S          V   S
     0 0 0 0 0 0 0 1 S S          V   S
```

Figure 17.3.4. Reduction process for intermediate blocks of the matrix in Figure 17.3.1. (a) Original form, (b) final form. (See text for explanation.)

```
(a)  0 0 0 1 0 0 0 0 S S          V   S
     0 0 0 0 1 0 0 0 S S          V   S
     0 0 0 0 0 1 0 0 S S          V   S
     0 0 0 0 0 0 1 0 S S          V   S
     0 0 0 0 0 0 0 1 S S          V   S
               Z Z Z D D          V   A
               Z Z Z D D          V   A

(b)  0 0 0 1 0 0 0 0 S S          V   S
     0 0 0 0 1 0 0 0 S S          V   S
     0 0 0 0 0 1 0 0 S S          V   S
     0 0 0 0 0 0 1 0 S S          V   S
     0 0 0 0 0 0 0 1 S S          V   S
               0 0 0 1 0          V   S
               0 0 0 0 1          V   S
```

Figure 17.3.5. Reduction process for the last (lower right) block of the matrix in Figure 17.3.1. (a) Original form, (b) final form. (See text for explanation.)

and (17.3.12), and the dependent variables. As described above it is essential that the `nb` boundary conditions at `k=1` involve the dependent variables referenced by the first `nb` columns of the `s` matrix. Thus, columns `j` of the `s` matrix can be ordered by the user in `difeq` to refer to derivatives with respect to the dependent variable `indexv(j)`.

The subroutine only attempts `itmax` correction cycles before returning, even if the solution has not converged. The parameters `conv, slowc, scalv` relate to convergence. Each inversion of the matrix produces corrections for `ne` variables at `m` mesh points. We want these to become vanishingly small as the iterations proceed, but we must define a measure for the size of corrections. This error "norm" is very problem specific, so the user might wish to rewrite this section of the code as appropriate. In the program below we compute a value for the average correction `err` by summing the absolute value of all corrections, weighted by a scale factor appropriate to each type of variable:

$$\texttt{err} = \frac{1}{\texttt{m} \times \texttt{ne}} \sum_{\texttt{k=1}}^{\texttt{m}} \sum_{\texttt{j=1}}^{\texttt{ne}} \frac{|\Delta Y\texttt{(j,k)}|}{\texttt{scalv(j)}} \tag{17.3.13}$$

When $\texttt{err} \leq \texttt{conv}$, the method has converged. Note that the user gets to supply an array `scalv` which measures the typical size of each variable.

Obviously, if `err` is large, we are far from a solution, and perhaps it is a bad idea to believe that the corrections generated from a first-order Taylor series are accurate. The number `slowc` modulates application of corrections. After each iteration we apply only a fraction of the corrections found by matrix inversion:

$$Y\texttt{(j,k)} \rightarrow Y\texttt{(j,k)} + \frac{\texttt{slowc}}{\max(\texttt{slowc},\texttt{err})} \Delta Y\texttt{(j,k)} \tag{17.3.14}$$

Thus, when $\texttt{err} > \texttt{slowc}$ only a fraction of the corrections are used, but when $\texttt{err} \leq \texttt{slowc}$ the entire correction gets applied.

The call statement also supplies `solvde` with the array `y(1:nyj,1:nyk)` containing the initial trial solution, and workspace arrays `c(1:nci,1:ncj,1:nck)`, `s(1:nsi,1:nsj)`. The array `c` is the blockbuster: It stores the unreduced elements of the matrix built up for the backsubstitution step. If there are `m` mesh points, then there will be `nck=m+1` blocks, each requiring `nci=ne` rows and `ncj=ne-nb+1` columns. Although large, this is small compared with $(\texttt{ne} \times \texttt{m})^2$ elements required for the whole matrix if we did not break it into blocks.

We now describe the workings of the user-supplied subroutine `difeq`. The parameters of the subroutine are given by

```
SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,ne,s,nsi,nsj,y,nyj,nyk)
```

The only information returned from `difeq` to `solvde` is the matrix of derivatives `s(i,j)`; all other arguments are input to `difeq` and should not be altered. `k` indicates the current mesh point, or block number. `k1,k2` label the first and last point in the mesh. If $\texttt{k} = \texttt{k1}$ or $\texttt{k} > \texttt{k2}$, the block involves the boundary conditions at the first or final points; otherwise the block acts on FDEs coupling variables at points `k-1`, `k`.

The convention on storing information into the array `s(i,j)` follows that used in equations (17.3.8), (17.3.10), and (17.3.12): Rows `i` label equations, columns `j` refer to derivatives with respect to dependent variables in the solution. Recall that each equation will depend on the `ne` dependent variables at either one or two points. Thus, `j` runs from `1` to either `ne` or `2*ne`. The column ordering for dependent variables at each point must agree with the list supplied in `indexv(j)`. Thus, for a block not at a boundary, the first column multiplies $\Delta Y(\texttt{l=indexv(1),k-1})$, and the column `ne+1` multiplies $\Delta Y(\texttt{l=indexv(1),k})$. `is1,isf` give the numbers of the starting and final *rows* that need to be filled in the `s` matrix for this block. `jsf` labels the column in which the difference equations $E_{j,k}$ of equations (17.3.3)–(17.3.5) are stored. Thus, $-\texttt{s(i,jsf)}$ is the vector on the right-hand side of the matrix. The reason for the minus sign is that `difeq` supplies the actual difference equation, $E_{j,k}$, not its negative. Note that `solvde` supplies a value for `jsf` such that the difference equation is put in the column *just after* all derivatives in the `s` matrix. Thus, `difeq` expects to find values entered into `s(i,j)` for rows $\texttt{is1} \leq \texttt{i} \leq \texttt{isf}$ and $1 \leq \texttt{j} \leq \texttt{jsf}$.

Finally, `s(1:nsi,1:nsj)` and `y(1:nyj,1:nyk)` supply `difeq` with storage for `s` and the solution variables `y` for this iteration. An example of how to use this routine is given in the next section.

Many ideas in the following code are due to Eggleton [1].

```
      SUBROUTINE solvde(itmax,conv,slowc,scalv,indexv,ne,nb,m,
     *     y,nyj,nyk,c,nci,ncj,nck,s,nsi,nsj)
      INTEGER itmax,m,nb,nci,ncj,nck,ne,nsi,nsj,
     *     nyj,nyk,indexv(nyj),NMAX
      REAL conv,slowc,c(nci,ncj,nck),s(nsi,nsj),
     *     scalv(nyj),y(nyj,nyk)
      PARAMETER (NMAX=10)           Largest expected value of ne.
C     USES bksub,difeq,pinvs,red
```

Driver routine for solution of two point boundary value problems by relaxation. `itmax` is the maximum number of iterations. `conv` is the convergence criterion (see text). `slowc` controls the fraction of corrections actually used after each iteration. `scalv(1:nyj)` contains typical sizes for each dependent variable, used to weight errors. `indexv(1:nyj)` lists the column ordering of variables used to construct the matrix `s` of derivatives. (The `nb` boundary conditions at the first mesh point must contain some dependence on the first `nb` variables listed in `indexv`.) The problem involves `ne` equations for `ne` adjustable dependent variables at each point. At the first mesh point there are `nb` boundary conditions. There are a total of `m` mesh points. `y(1:nyj,1:nyk)` is the two-dimensional array that contains the initial guess for all the dependent variables at each mesh point. On each iteration, it is updated by the calculated correction. The arrays `c(1:nci,1:ncj,1:nck)`, `s(1:nsi,1:nsj)` supply dummy storage used by the relaxation code; the minimum dimensions must satisfy: `nci=ne`, `ncj=ne-nb+1`, `nck=m+1`, `nsi=ne`, `nsj=2*ne+1`.

```
      INTEGER ic1,ic2,ic3,ic4,it,j,j1,j2,j3,j4,j5,j6,j7,j8,
     *     j9,jc1,jcf,jv,k,k1,k2,km,kp,nvars,kmax(NMAX)
      REAL err,errj,fac,vmax,vz,ermax(NMAX)
      k1=1                          Set up row and column markers.
      k2=m
      nvars=ne*m
      j1=1
      j2=nb
      j3=nb+1
      j4=ne
      j5=j4+j1
      j6=j4+j2
      j7=j4+j3
      j8=j4+j4
      j9=j8+j1
      ic1=1
      ic2=ne-nb
      ic3=ic2+1
      ic4=ne
      jc1=1
      jcf=ic3
      do 16 it=1,itmax              Primary iteration loop.
        k=k1                        Boundary conditions at first point.
        call difeq(k,k1,k2,j9,ic3,ic4,indexv,ne,s,nsi,nsj,y,nyj,nyk)
        call pinvs(ic3,ic4,j5,j9,jc1,k1,c,nci,ncj,nck,s,nsi,nsj)
        do 11 k=k1+1,k2             Finite difference equations at all point pairs.
          kp=k-1
          call difeq(k,k1,k2,j9,ic1,ic4,indexv,ne,s,nsi,nsj,y,nyj,nyk)
          call red(ic1,ic4,j1,j2,j3,j4,j9,ic3,jc1,jcf,kp,
     *         c,nci,ncj,nck,s,nsi,nsj)
          call pinvs(ic1,ic4,j3,j9,jc1,k,c,nci,ncj,nck,s,nsi,nsj)
        enddo 11
        k=k2+1                      Final boundary conditions.
        call difeq(k,k1,k2,j9,ic1,ic2,indexv,ne,s,nsi,nsj,y,nyj,nyk)
        call red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jc1,jcf,k2,
     *       c,nci,ncj,nck,s,nsi,nsj)
        call pinvs(ic1,ic2,j7,j9,jcf,k2+1,c,nci,ncj,nck,s,nsi,nsj)
```

```
        call bksub(ne,nb,jcf,k1,k2,c,nci,ncj,nck)        Backsubstitution.
        err=0.
        do 13 j=1,ne                                     Convergence check, accumulate average error.
            jv=indexv(j)
            errj=0.
            km=0
            vmax=0.
            do 12 k=k1,k2                                Find point with largest error, for each dependent variable.
                vz=abs(c(jv,1,k))
                if(vz.gt.vmax) then
                    vmax=vz
                    km=k
                endif
                errj=errj+vz
            enddo 12
            err=err+errj/scalv(j)                        Note weighting for each dependent variable.
            ermax(j)=c(jv,1,km)/scalv(j)
            kmax(j)=km
        enddo 13
        err=err/nvars
        fac=slowc/max(slowc,err)                         Reduce correction applied when error is large.
        do 15 j=1,ne                                     Apply corrections.
            jv=indexv(j)
            do 14 k=k1,k2
                y(j,k)=y(j,k)-fac*c(jv,1,k)
            enddo 14
        enddo 15
        write(*,100) it,err,fac                          Summary of corrections for this step. Point with largest
        if(err.lt.conv) return                               error for each variable can be monitored by writ-
    enddo 16                                                 ing out kmax and ermax.
    pause 'itmax exceeded in solvde'                     Convergence failed.
100 format(1x,i4,2f12.6)
    return
    END
    SUBROUTINE bksub(ne,nb,jf,k1,k2,c,nci,ncj,nck)
    INTEGER jf,k1,k2,nb,nci,ncj,nck,ne
    REAL c(nci,ncj,nck)
      Backsubstitution, used internally by solvde.
    INTEGER i,im,j,k,kp,nbf
    REAL xx
    nbf=ne-nb
    im=1
    do 13 k=k2,k1,-1                                     Use recurrence relations to eliminate remaining dependences.
        if (k.eq.k1) im=nbf+1                            Special handling of first point.
        kp=k+1
        do 12 j=1,nbf
            xx=c(j,jf,kp)
            do 11 i=im,ne
                c(i,jf,k)=c(i,jf,k)-c(i,j,k)*xx
            enddo 11
        enddo 12
    enddo 13
    do 16 k=k1,k2                                        Reorder corrections to be in column 1.
        kp=k+1
        do 14 i=1,nb
            c(i,1,k)=c(i+nbf,jf,k)
        enddo 14
        do 15 i=1,nbf
            c(i+nb,1,k)=c(i,jf,kp)
        enddo 15
    enddo 16
    return
    END
```

```
SUBROUTINE pinvs(ie1,ie2,je1,jsf,jc1,k,c,nci,ncj,nck,s,nsi,nsj)
INTEGER ie1,ie2,jc1,je1,jsf,k,nci,ncj,nck,nsi,nsj,NMAX
REAL c(nci,ncj,nck),s(nsi,nsj)
PARAMETER (NMAX=10)
    Diagonalize the square subsection of the s matrix, and store the recursion coefficients in
    c; used internally by solvde.
INTEGER i,icoff,id,ipiv,irow,j,jcoff,je2,jp,jpiv,js1,indxr(NMAX)
REAL big,dum,piv,pivinv,pscl(NMAX)
je2=je1+ie2-ie1
js1=je2+1
do 12 i=ie1,ie2                 Implicit pivoting, as in §2.1.
    big=0.
    do 11 j=je1,je2
        if(abs(s(i,j)).gt.big) big=abs(s(i,j))
    enddo 11
    if(big.eq.0.) pause 'singular matrix, row all 0 in pinvs'
    pscl(i)=1./big
    indxr(i)=0
enddo 12
do 18 id=ie1,ie2
    piv=0.
    do 14 i=ie1,ie2             Find pivot element.
        if(indxr(i).eq.0) then
            big=0.
            do 13 j=je1,je2
                if(abs(s(i,j)).gt.big) then
                    jp=j
                    big=abs(s(i,j))
                endif
            enddo 13
            if(big*pscl(i).gt.piv) then
                ipiv=i
                jpiv=jp
                piv=big*pscl(i)
            endif
        endif
    enddo 14
    if(s(ipiv,jpiv).eq.0.) pause 'singular matrix in pinvs'
    indxr(ipiv)=jpiv            In place reduction. Save column ordering.
    pivinv=1./s(ipiv,jpiv)
    do 15 j=je1,jsf             Normalize pivot row.
        s(ipiv,j)=s(ipiv,j)*pivinv
    enddo 15
    s(ipiv,jpiv)=1.
    do 17 i=ie1,ie2             Reduce nonpivot elements in column.
        if(indxr(i).ne.jpiv) then
            if(s(i,jpiv).ne.0.) then
                dum=s(i,jpiv)
                do 16 j=je1,jsf
                    s(i,j)=s(i,j)-dum*s(ipiv,j)
                enddo 16
                s(i,jpiv)=0.
            endif
        endif
    enddo 17
enddo 18
jcoff=jc1-js1                   Sort and store unreduced coefficients.
icoff=ie1-je1
do 21 i=ie1,ie2
    irow=indxr(i)+icoff
    do 19 j=js1,jsf
        c(irow,j+jcoff,k)=s(i,j)
    enddo 19
enddo 21
```

```
      return
      END


      SUBROUTINE red(iz1,iz2,jz1,jz2,jm1,jm2,jmf,ic1,jc1,jcf,kc,
     *     c,nci,ncj,nck,s,nsi,nsj)
      INTEGER ic1,iz1,iz2,jc1,jcf,jm1,jm2,jmf,jz1,jz2,kc,nci,ncj,
     *     nck,nsi,nsj
      REAL c(nci,ncj,nck),s(nsi,nsj)
        Reduce columns jz1-jz2 of the s matrix, using previous results as stored in the c matrix.
        Only columns jm1-jm2,jmf are affected by the prior results. red is used internally by
        solvde.
      INTEGER i,ic,j,l,loff
      REAL vx
      loff=jc1-jm1
      ic=ic1
      do 14 j=jz1,jz2                Loop over columns to be zeroed.
        do 12 l=jm1,jm2              Loop over columns altered.
          vx=c(ic,l+loff,kc)
          do 11 i=iz1,iz2            Loop over rows.
            s(i,l)=s(i,l)-s(i,j)*vx
          enddo 11
        enddo 12
        vx=c(ic,jcf,kc)
        do 13 i=iz1,iz2              Plus final element.
          s(i,jmf)=s(i,jmf)-s(i,j)*vx
        enddo 13
        ic=ic+1
      enddo 14
      return
      END
```

"Algebraically Difficult" Sets of Differential Equations

Relaxation methods allow you to take advantage of an additional opportunity that, while not obvious, can speed up some calculations enormously. It is not necessary that the set of variables $y_{j,k}$ correspond exactly with the dependent variables of the original differential equations. They can be related to those variables through algebraic equations. Obviously, it is necessary only that the solution variables allow us to *evaluate* the functions $y, g, \mathbf{B}, \mathbf{C}$ that are used to construct the FDEs from the ODEs. In some problems g depends on functions of y that are known only implicitly, so that iterative solutions are necessary to evaluate functions in the ODEs. Often one can dispense with this "internal" nonlinear problem by defining a new set of variables from which both y, g and the boundary conditions can be obtained directly. A typical example occurs in physical problems where the equations require solution of a complex equation of state that can be expressed in more convenient terms using variables other than the original dependent variables in the ODE. While this approach is analogous to performing an *analytic* change of variables directly on the original ODEs, such an analytic transformation might be prohibitively complicated. The change of variables in the relaxation method is easy and requires no analytic manipulations.

CITED REFERENCES AND FURTHER READING:

Eggleton, P.P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364. [1]

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell).

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

17.4 A Worked Example: Spheroidal Harmonics

The best way to understand the algorithms of the previous sections is to see them employed to solve an actual problem. As a sample problem, we have selected the computation of spheroidal harmonics. (The more common name is spheroidal angle functions, but we prefer the explicit reminder of the kinship with spherical harmonics.) We will show how to find spheroidal harmonics, first by the method of relaxation (§17.3), and then by the methods of shooting (§17.1) and shooting to a fitting point (§17.2).

Spheroidal harmonics typically arise when certain partial differential equations are solved by separation of variables in spheroidal coordinates. They satisfy the following differential equation on the interval $-1 \le x \le 1$:

$$\frac{d}{dx}\left[(1-x^2)\frac{dS}{dx}\right] + \left(\lambda - c^2x^2 - \frac{m^2}{1-x^2}\right)S = 0 \tag{17.4.1}$$

Here m is an integer, c is the "oblateness parameter," and λ is the eigenvalue. Despite the notation, c^2 can be positive or negative. For $c^2 > 0$ the functions are called "prolate," while if $c^2 < 0$ they are called "oblate." The equation has singular points at $x = \pm 1$ and is to be solved subject to the boundary conditions that the solution be regular at $x = \pm 1$. Only for certain values of λ, the eigenvalues, will this be possible.

If we consider first the spherical case, where $c = 0$, we recognize the differential equation for Legendre functions $P_n^m(x)$. In this case the eigenvalues are $\lambda_{mn} = n(n+1)$, $n = m, m+1, \ldots$. The integer n labels successive eigenvalues for fixed m: When $n = m$ we have the lowest eigenvalue, and the corresponding eigenfunction has no nodes in the interval $-1 < x < 1$; when $n = m+1$ we have the next eigenvalue, and the eigenfunction has one node inside $(-1, 1)$; and so on.

A similar situation holds for the general case $c^2 \neq 0$. We write the eigenvalues of (17.4.1) as $\lambda_{mn}(c)$ and the eigenfunctions as $S_{mn}(x; c)$. For fixed m, $n = m, m+1, \ldots$ labels the successive eigenvalues.

The computation of $\lambda_{mn}(c)$ and $S_{mn}(x; c)$ traditionally has been quite difficult. Complicated recurrence relations, power series expansions, etc., can be found in references [1-3]. Cheap computing makes evaluation by direct solution of the differential equation quite feasible.

The first step is to investigate the behavior of the solution near the singular points $x = \pm 1$. Substituting a power series expansion of the form

$$S = (1 \pm x)^\alpha \sum_{k=0}^{\infty} a_k (1 \pm x)^k \tag{17.4.2}$$

in equation (17.4.1), we find that the regular solution has $\alpha = m/2$. (Without loss of generality we can take $m \ge 0$ since $m \to -m$ is a symmetry of the equation.) We get an equation that is numerically more tractable if we factor out this behavior. Accordingly we set

$$S = (1-x^2)^{m/2} y \tag{17.4.3}$$

We then find from (17.4.1) that y satisfies the equation

$$(1-x^2)\frac{d^2y}{dx^2} - 2(m+1)x\frac{dy}{dx} + (\mu - c^2x^2)y = 0 \tag{17.4.4}$$

17.4 A Worked Example: Spheroidal Harmonics

The best way to understand the algorithms of the previous sections is to see them employed to solve an actual problem. As a sample problem, we have selected the computation of spheroidal harmonics. (The more common name is spheroidal angle functions, but we prefer the explicit reminder of the kinship with spherical harmonics.) We will show how to find spheroidal harmonics, first by the method of relaxation (§17.3), and then by the methods of shooting (§17.1) and shooting to a fitting point (§17.2).

Spheroidal harmonics typically arise when certain partial differential equations are solved by separation of variables in spheroidal coordinates. They satisfy the following differential equation on the interval $-1 \le x \le 1$:

$$\frac{d}{dx}\left[(1-x^2)\frac{dS}{dx}\right] + \left(\lambda - c^2x^2 - \frac{m^2}{1-x^2}\right)S = 0 \tag{17.4.1}$$

Here m is an integer, c is the "oblateness parameter," and λ is the eigenvalue. Despite the notation, c^2 can be positive or negative. For $c^2 > 0$ the functions are called "prolate," while if $c^2 < 0$ they are called "oblate." The equation has singular points at $x = \pm 1$ and is to be solved subject to the boundary conditions that the solution be regular at $x = \pm 1$. Only for certain values of λ, the eigenvalues, will this be possible.

If we consider first the spherical case, where $c = 0$, we recognize the differential equation for Legendre functions $P_n^m(x)$. In this case the eigenvalues are $\lambda_{mn} = n(n+1)$, $n = m, m+1, \ldots$. The integer n labels successive eigenvalues for fixed m: When $n = m$ we have the lowest eigenvalue, and the corresponding eigenfunction has no nodes in the interval $-1 < x < 1$; when $n = m+1$ we have the next eigenvalue, and the eigenfunction has one node inside $(-1, 1)$; and so on.

A similar situation holds for the general case $c^2 \neq 0$. We write the eigenvalues of (17.4.1) as $\lambda_{mn}(c)$ and the eigenfunctions as $S_{mn}(x; c)$. For fixed m, $n = m, m+1, \ldots$ labels the successive eigenvalues.

The computation of $\lambda_{mn}(c)$ and $S_{mn}(x; c)$ traditionally has been quite difficult. Complicated recurrence relations, power series expansions, etc., can be found in references [1-3]. Cheap computing makes evaluation by direct solution of the differential equation quite feasible.

The first step is to investigate the behavior of the solution near the singular points $x = \pm 1$. Substituting a power series expansion of the form

$$S = (1 \pm x)^\alpha \sum_{k=0}^{\infty} a_k (1 \pm x)^k \tag{17.4.2}$$

in equation (17.4.1), we find that the regular solution has $\alpha = m/2$. (Without loss of generality we can take $m \ge 0$ since $m \to -m$ is a symmetry of the equation.) We get an equation that is numerically more tractable if we factor out this behavior. Accordingly we set

$$S = (1-x^2)^{m/2} y \tag{17.4.3}$$

We then find from (17.4.1) that y satisfies the equation

$$(1-x^2)\frac{d^2y}{dx^2} - 2(m+1)x\frac{dy}{dx} + (\mu - c^2x^2)y = 0 \tag{17.4.4}$$

where

$$\mu \equiv \lambda - m(m+1) \tag{17.4.5}$$

Both equations (17.4.1) and (17.4.4) are invariant under the replacement $x \to -x$. Thus the functions S and y must also be invariant, except possibly for an overall scale factor. (Since the equations are linear, a constant multiple of a solution is also a solution.) Because the solutions will be normalized, the scale factor can only be ± 1. If $n-m$ is odd, there are an odd number of zeros in the interval $(-1,1)$. Thus we must choose the antisymmetric solution $y(-x) = -y(x)$ which has a zero at $x = 0$. Conversely, if $n-m$ is even we must have the symmetric solution. Thus

$$y_{mn}(-x) = (-1)^{n-m} y_{mn}(x) \tag{17.4.6}$$

and similarly for S_{mn}.

The boundary conditions on (17.4.4) require that y be regular at $x = \pm 1$. In other words, near the endpoints the solution takes the form

$$y = a_0 + a_1(1-x^2) + a_2(1-x^2)^2 + \ldots \tag{17.4.7}$$

Substituting this expansion in equation (17.4.4) and letting $x \to 1$, we find that

$$a_1 = -\frac{\mu - c^2}{4(m+1)} a_0 \tag{17.4.8}$$

Equivalently,

$$y'(1) = \frac{\mu - c^2}{2(m+1)} y(1) \tag{17.4.9}$$

A similar equation holds at $x = -1$ with a minus sign on the right-hand side. The irregular solution has a different relation between function and derivative at the endpoints.

Instead of integrating the equation from -1 to 1, we can exploit the symmetry (17.4.6) to integrate from 0 to 1. The boundary condition at $x = 0$ is

$$\begin{aligned} y(0) &= 0, \quad n-m \text{ odd} \\ y'(0) &= 0, \quad n-m \text{ even} \end{aligned} \tag{17.4.10}$$

A third boundary condition comes from the fact that any constant multiple of a solution y is a solution. We can thus *normalize* the solution. We adopt the normalization that the function S_{mn} has the same limiting behavior as P_n^m at $x = 1$:

$$\lim_{x\to 1} (1-x^2)^{-m/2} S_{mn}(x;c) = \lim_{x\to 1} (1-x^2)^{-m/2} P_n^m(x) \tag{17.4.11}$$

Various normalization conventions in the literature are tabulated by Flammer [1].

Imposing three boundary conditions for the second-order equation (17.4.4) turns it into an eigenvalue problem for λ or equivalently for μ. We write it in the standard form by setting

$$y_1 = y \tag{17.4.12}$$
$$y_2 = y' \tag{17.4.13}$$
$$y_3 = \mu \tag{17.4.14}$$

Then

$$y_1' = y_2 \tag{17.4.15}$$
$$y_2' = \frac{1}{1-x^2}\left[2x(m+1)y_2 - (y_3 - c^2x^2)y_1\right] \tag{17.4.16}$$
$$y_3' = 0 \tag{17.4.17}$$

The boundary condition at $x = 0$ in this notation is

$$\begin{aligned} y_1 &= 0, \quad n-m \text{ odd} \\ y_2 &= 0, \quad n-m \text{ even} \end{aligned} \tag{17.4.18}$$

At $x = 1$ we have two conditions:

$$y_2 = \frac{y_3 - c^2}{2(m+1)} y_1 \tag{17.4.19}$$
$$y_1 = \lim_{x\to 1}(1-x^2)^{-m/2} P_n^m(x) = \frac{(-1)^m (n+m)!}{2^m m!(n-m)!} \equiv \gamma \tag{17.4.20}$$

We are now ready to illustrate the use of the methods of previous sections on this problem.

Relaxation

If we just want a few isolated values of λ or S, shooting is probably the quickest method. However, if we want values for a large sequence of values of c, relaxation is better. Relaxation rewards a good initial guess with rapid convergence, and the previous solution should be a good initial guess if c is changed only slightly.

For simplicity, we choose a uniform grid on the interval $0 \le x \le 1$. For a total of M mesh points, we have

$$h = \frac{1}{M-1} \tag{17.4.21}$$
$$x_k = (k-1)h, \qquad k = 1, 2, \ldots, M \tag{17.4.22}$$

At interior points $k = 2, 3, \ldots, M$, equation (17.4.15) gives

$$E_{1,k} = y_{1,k} - y_{1,k-1} - \frac{h}{2}(y_{2,k} + y_{2,k-1}) \tag{17.4.23}$$

Equation (17.4.16) gives

$$E_{2,k} = y_{2,k} - y_{2,k-1} - \beta_k \times \left[\frac{(x_k + x_{k-1})(m+1)(y_{2,k} + y_{2,k-1})}{2} - \alpha_k \frac{(y_{1,k} + y_{1,k-1})}{2} \right] \tag{17.4.24}$$

where

$$\alpha_k = \frac{y_{3,k} + y_{3,k-1}}{2} - \frac{c^2 (x_k + x_{k-1})^2}{4} \tag{17.4.25}$$

$$\beta_k = \frac{h}{1 - \frac{1}{4}(x_k + x_{k-1})^2} \tag{17.4.26}$$

Finally, equation (17.4.17) gives

$$E_{3,k} = y_{3,k} - y_{3,k-1} \tag{17.4.27}$$

Now recall that the matrix of partial derivatives $S_{i,j}$ of equation (17.3.8) is defined so that i labels the equation and j the variable. In our case, j runs from 1 to 3 for y_j at $k-1$ and from 4 to 6 for y_j at k. Thus equation (17.4.23) gives

$$\begin{aligned} S_{1,1} &= -1, & S_{1,2} &= -\frac{h}{2}, & S_{1,3} &= 0 \\ S_{1,4} &= 1, & S_{1,5} &= -\frac{h}{2}, & S_{1,6} &= 0 \end{aligned} \tag{17.4.28}$$

Similarly equation (17.4.24) yields

$$\begin{aligned} S_{2,1} &= \alpha_k \beta_k / 2, & S_{2,2} &= -1 - \beta_k (x_k + x_{k-1})(m+1)/2, \\ S_{2,3} &= \beta_k (y_{1,k} + y_{1,k-1})/4 & S_{2,4} &= S_{2,1}, \\ S_{2,5} &= 2 + S_{2,2}, & S_{2,6} &= S_{2,3} \end{aligned} \tag{17.4.29}$$

while from equation (17.4.27) we find

$$\begin{aligned} S_{3,1} &= 0, & S_{3,2} &= 0, & S_{3,3} &= -1 \\ S_{3,4} &= 0, & S_{3,5} &= 0, & S_{3,6} &= 1 \end{aligned} \tag{17.4.30}$$

At $x = 0$ we have the boundary condition

$$E_{3,1} = \begin{cases} y_{1,1}, & n - m \text{ odd} \\ y_{2,1}, & n - m \text{ even} \end{cases} \tag{17.4.31}$$

Recall the convention adopted in the `solvde` routine that for one boundary condition at $k = 1$ only $S_{3,j}$ can be nonzero. Also, j takes on the values 4 to 6 since the boundary condition involves only y_k, not y_{k-1}. Accordingly, the only nonzero values of $S_{3,j}$ at $x = 0$ are

$$\begin{aligned} S_{3,4} &= 1, & n - m \text{ odd} \\ S_{3,5} &= 1, & n - m \text{ even} \end{aligned} \tag{17.4.32}$$

At $x = 1$ we have

$$E_{1,M+1} = y_{2,M} - \frac{y_{3,M} - c^2}{2(m+1)} y_{1,M} \tag{17.4.33}$$

$$E_{2,M+1} = y_{1,M} - \gamma \tag{17.4.34}$$

Thus

$$S_{1,4} = -\frac{y_{3,M} - c^2}{2(m+1)}, \qquad S_{1,5} = 1, \qquad S_{1,6} = -\frac{y_{1,M}}{2(m+1)} \tag{17.4.35}$$

$$S_{2,4} = 1, \qquad S_{2,5} = 0, \qquad S_{2,6} = 0 \tag{17.4.36}$$

Here now is the sample program that implements the above algorithm. We need a main program, `sfroid`, that calls the routine `solvde`, and we must supply the subroutine `difeq` called by `solvde`. For simplicity we choose an equally spaced mesh of `m` = 41 points, that is, $h = .025$. As we shall see, this gives good accuracy for the eigenvalues up to moderate values of $n - m$.

Since the boundary condition at $x = 0$ does not involve y_1 if $n - m$ is even, we have to use the `indexv` feature of `solvde`. Recall that the value of `indexv(j)` describes which column of `s(i,j)` the variable `y(j)` has been put in. If $n - m$ is even, we need to interchange the columns for y_1 and y_2 so that there is not a zero pivot element in `s(i,j)`.

The program prompts for values of m and n. It then computes an initial guess for y based on the Legendre function P_n^m. It next prompts for c^2, solves for y, prompts for c^2, solves for y using the previous values as an initial guess, and so on.

```
      PROGRAM sfroid
      INTEGER NE,M,NB,NCI,NCJ,NCK,NSI,NSJ,NYJ,NYK
      COMMON /sfrcom/ x,h,mm,n,c2,anorm            Communicates with difeq.
      PARAMETER (NE=3,M=41,NB=1,NCI=NE,NCJ=NE-NB+1,NCK=M+1,NSI=NE,
     *     NSJ=2*NE+1,NYJ=NE,NYK=M)
C     USES plgndr,solvde
        Sample program using solvde. Computes eigenvalues of spheroidal harmonics S_mn(x;c)
        for m ≥ 0 and n ≥ m. In the program, m is mm, c^2 is c2, and γ of equation (17.4.20)
        is anorm.
      INTEGER i,itmax,k,mm,n,indexv(NE)
      REAL anorm,c2,conv,deriv,fac1,fac2,h,q1,slowc,
     *     c(NCI,NCJ,NCK),s(NSI,NSJ),scalv(NE),x(M),y(NE,M),plgndr
      itmax=100
      conv=5.e-6
      slowc=1.
      h=1./(M-1)
      c2=0.
      write(*,*)'ENTER M,N'
      read(*,*)mm,n
      if(mod(n+mm,2).eq.1)then                 No interchanges necessary.
        indexv(1)=1
        indexv(2)=2
        indexv(3)=3
      else                                     Interchange y_1 and y_2.
        indexv(1)=2
        indexv(2)=1
        indexv(3)=3
      endif
      anorm=1.                                 Compute γ.
```

```
      if(mm.NE.0)then
        q1=n
        do 11 i=1,mm
          anorm=-.5*anorm*(n+i)*(q1/i)
          q1=q1-1.
        enddo 11
      endif
      do 12 k=1,M-1                                   Initial guess.
        x(k)=(k-1)*h
        fac1=1.-x(k)**2
        fac2=fac1**(-mm/2.)
        y(1,k)=plgndr(n,mm,x(k))*fac2                 P_n^m from §6.8.
        deriv=-((n-mm+1)*plgndr(n+1,mm,x(k))-(n+1)*
     *       x(k)*plgndr(n,mm,x(k)))/fac1             Derivative of P_n^m from a recurrence re-
        y(2,k)=mm*x(k)*y(1,k)/fac1+deriv*fac2             lation.
        y(3,k)=n*(n+1)-mm*(mm+1)
      enddo 12
      x(M)=1.                                         Initial guess at x = 1 done separately.
      y(1,M)=anorm
      y(3,M)=n*(n+1)-mm*(mm+1)
      y(2,M)=(y(3,M)-c2)*y(1,M)/(2.*(mm+1.))
      scalv(1)=abs(anorm)
      scalv(2)=max(abs(anorm),y(2,M))
      scalv(3)=max(1.,y(3,M))
1     continue
      write (*,*) 'ENTER C**2 OR 999 TO END'
      read (*,*) c2
      if (c2.eq.999.) stop
      call solvde(itmax,conv,slowc,scalv,indexv,NE,NB,M,y,NYJ,NYK,
     *     c,NCI,NCJ,NCK,s,NSI,NSJ)
      write (*,*) ' M = ',mm,' N = ',n,
     *     ' C**2 = ',c2,' LAMBDA = ',y(3,1)+mm*(mm+1)
      goto 1                                          for another value of c^2.
      END


      SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,ne,s,nsi,nsj,y,nyj,nyk)
      INTEGER is1,isf,jsf,k,k1,k2,ne,nsi,nsj,nyj,nyk,indexv(nyj),M
      REAL s(nsi,nsj),y(nyj,nyk)
      COMMON /sfrcom/ x,h,mm,n,c2,anorm
      PARAMETER (M=41)
        Returns matrix s(i,j) for solvde.
      INTEGER mm,n
      REAL anorm,c2,h,temp,temp2,x(M)
      if(k.eq.k1) then                    Boundary condition at first point.
        if(mod(n+mm,2).eq.1)then
          s(3,3+indexv(1))=1.             Equation (17.4.32).
          s(3,3+indexv(2))=0.
          s(3,3+indexv(3))=0.
          s(3,jsf)=y(1,1)                 Equation (17.4.31).
        else
          s(3,3+indexv(1))=0.             Equation (17.4.32).
          s(3,3+indexv(2))=1.
          s(3,3+indexv(3))=0.
          s(3,jsf)=y(2,1)                 Equation (17.4.31).
        endif
      else if(k.gt.k2) then               Boundary conditions at last point.
        s(1,3+indexv(1))=-(y(3,M)-c2)/(2.*(mm+1.))        Equation (17.4.35).
        s(1,3+indexv(2))=1.
        s(1,3+indexv(3))=-y(1,M)/(2.*(mm+1.))
        s(1,jsf)=y(2,M)-(y(3,M)-c2)*y(1,M)/(2.*(mm+1.))   Equation (17.4.33).
        s(2,3+indexv(1))=1.               Equation (17.4.36).
        s(2,3+indexv(2))=0.
```

```
        s(2,3+indexv(3))=0.
        s(2,jsf)=y(1,M)-anorm           Equation (17.4.34).
    else                                Interior point.
        s(1,indexv(1))=-1.              Equation (17.4.28).
        s(1,indexv(2))=-.5*h
        s(1,indexv(3))=0.
        s(1,3+indexv(1))=1.
        s(1,3+indexv(2))=-.5*h
        s(1,3+indexv(3))=0.
        temp=h/(1.-(x(k)+x(k-1))**2*.25)
        temp2=.5*(y(3,k)+y(3,k-1))-c2*.25*(x(k)+x(k-1))**2
        s(2,indexv(1))=temp*temp2*.5                          Equation (17.4.29).
        s(2,indexv(2))=-1.-.5*temp*(mm+1.)*(x(k)+x(k-1))
        s(2,indexv(3))=.25*temp*(y(1,k)+y(1,k-1))
        s(2,3+indexv(1))=s(2,indexv(1))
        s(2,3+indexv(2))=2.+s(2,indexv(2))
        s(2,3+indexv(3))=s(2,indexv(3))
        s(3,indexv(1))=0.                                     Equation (17.4.30).
        s(3,indexv(2))=0.
        s(3,indexv(3))=-1.
        s(3,3+indexv(1))=0.
        s(3,3+indexv(2))=0.
        s(3,3+indexv(3))=1.
        s(1,jsf)=y(1,k)-y(1,k-1)-.5*h*(y(2,k)+y(2,k-1))       Equation (17.4.23).
        s(2,jsf)=y(2,k)-y(2,k-1)-temp*((x(k)+x(k-1))          Equation (17.4.24).
*            *.5*(mm+1.)*(y(2,k)+y(2,k-1))-temp2*
*            .5*(y(1,k)+y(1,k-1)))
        s(3,jsf)=y(3,k)-y(3,k-1)                              Equation (17.4.27).
    endif
    return
    END
```

You can run the program and check it against values of $\lambda_{mn}(c)$ given in the tables at the back of Flammer's book [1] or in Table 21.1 of Abramowitz and Stegun [2]. Typically it converges in about 3 iterations. The table below gives a few comparisons.

Selected Output of `sfroid`

m	n	c^2	λ_{exact}	$\lambda_{\texttt{sfroid}}$
2	2	0.1	6.01427	6.01427
		1.0	6.14095	6.14095
		4.0	6.54250	6.54253
2	5	1.0	30.4361	30.4372
		16.0	36.9963	37.0135
4	11	-1.0	131.560	131.554

Shooting

To solve the same problem via shooting (§17.1), we supply a subroutine `derivs` that implements equations (17.4.15)–(17.4.17). We will integrate the equations over the range $-1 \le x \le 0$. We provide the subroutine `load` which sets the eigenvalue y_3 to its current best estimate, `v(1)`. It also sets the boundary values of y_1 and

y_2 using equations (17.4.20) and (17.4.19) (with a minus sign corresponding to $x = -1$). Note that the boundary condition is actually applied a distance dx from the boundary to avoid having to evaluate y_2' right on the boundary. The subroutine score follows from equation (17.4.18).

```
      PROGRAM sphoot
        Sample program using shoot. Computes eigenvalues of spheroidal harmonics S_mn(x;c) for
        m ≥ 0 and n ≥ m. Be sure that routine funcv for newt is provided by shoot (§17.1).
      INTEGER i,m,n,nvar,N2
      PARAMETER (N2=1)
      REAL c2,dx,gamma,q1,x1,x2,v(N2)
      LOGICAL check
      COMMON /sphcom/ c2,gamma,dx,m,n          Communicates with load, score, and derivs.
      COMMON /caller/ x1,x2,nvar               Communicates with shoot.
C     USES newt
      dx=1.e-4                                 Avoid evaluating derivatives exactly at x = −1.
      nvar=3                                   Number of equations.
1     write(*,*) 'input m,n,c-squared (999 to end)'
      read(*,*) m,n,c2
      if (c2.eq.999.) stop
      if ((n.lt.m).or.(m.lt.0)) goto 1
      gamma=1.0                                Compute γ of equation (17.4.20).
      q1=n
      do 11 i=1,m
        gamma=-0.5*gamma*(n+i)*(q1/i)
        q1=q1-1.0
      enddo 11
      v(1)=n*(n+1)-m*(m+1)+c2/2.0              Initial guess for eigenvalue.
      x1=-1.0+dx                               Set range of integration.
      x2=0.0
      call newt(v,N2,check)                    Find v that zeros function f in score.
      if(check)then
        write(*,*)'shoot failed; bad initial guess'
      else
        write(*,'(1x,t6,a)') 'mu(m,n)'
        write(*,'(1x,f12.6)') v(1)
        goto 1
      endif
      END

      SUBROUTINE load(x1,v,y)
      INTEGER m,n
      REAL c2,dx,gamma,x1,y1,v(1),y(3)
      COMMON /sphcom/ c2,gamma,dx,m,n
        Supplies starting values for integration at x = −1 + dx.
      y(3)=v(1)
      if(mod(n-m,2).eq.0)then
        y1=gamma
      else
        y1=-gamma
      endif
      y(2)=-(y(3)-c2)*y1/(2*(m+1))
      y(1)=y1+y(2)*dx
      return
      END

      SUBROUTINE score(x2,y,f)
      INTEGER m,n
      REAL c2,dx,gamma,x2,f(1),y(3)
      COMMON /sphcom/ c2,gamma,dx,m,n
        Tests whether boundary condition at x = 0 is satisfied.
      if (mod(n-m,2).eq.0) then
        f(1)=y(2)
```

```
      else
          f(1)=y(1)
      endif
      return
      END

      SUBROUTINE derivs(x,y,dydx)
      INTEGER m,n
      REAL c2,dx,gamma,x,dydx(3),y(3)
      COMMON /sphcom/ c2,gamma,dx,m,n
         Evaluates derivatives for odeint.
      dydx(1)=y(2)
      dydx(2)=(2.0*x*(m+1.0)*y(2)-(y(3)-c2*x*x)*y(1))/(1.0-x*x)
      dydx(3)=0.0
      return
      END
```

Shooting to a Fitting Point

For variety we illustrate `shootf` from §17.2 by integrating over the whole range $-1 + dx \le x \le 1 - dx$, with the fitting point chosen to be at $x = 0$. The routine `derivs` is identical to the one for `shoot`. Now, however, there are two load routines. The routine `load1` for $x = -1$ is essentially identical to `load` above. At $x = 1$, `load2` sets the function value y_1 and the eigenvalue y_3 to their best current estimates, `v2(1)` and `v2(2)`, respectively. If you quite sensibly make your initial guess of the eigenvalue the same in the two intervals, then `v1(1)` will stay equal to `v2(2)` during the iteration. The subroutine `score` simply checks whether all three function values match at the fitting point.

```
      PROGRAM sphfpt
         Sample program using shootf. Computes eigenvalues of spheroidal harmonics S_mn(x;c)
         for m ≥ 0 and n ≥ m. Be sure that routine funcv for newt is provided by shootf (§17.2).
         The routine derivs is the same as for sphoot.
      INTEGER i,m,n,nvar,nn2,N1,N2,NTOT
      REAL DXX
      PARAMETER (N1=2,N2=1,NTOT=N1+N2,DXX=1.e-4)
      REAL c2,dx,gamma,q1,x1,x2,xf,v1(N2),v2(N1),v(NTOT)
      LOGICAL check
      COMMON /sphcom/ c2,gamma,dx,m,n
        Communicates with load1, load2, score, and derivs.
      COMMON /caller/ x1,x2,xf,nvar,nn2      Communicates with shootf.
      EQUIVALENCE (v1(1),v(1)),(v2(1),v(N2+1))
C     USES newt
      nvar=NTOT                              Number of equations.
      nn2=N2
      dx=DXX                                 Avoid evaluating derivatives exactly at x = ±1.
1     write(*,*) 'input m,n,c-squared (999 to end)'
      read(*,*) m,n,c2
      if (c2.eq.999.) stop
      if ((n.lt.m).or.(m.lt.0)) goto 1
      gamma=1.0                              Compute γ of equation (17.4.20).
      q1=n
      do 11 i=1,m
          gamma=-0.5*gamma*(n+i)*(q1/i)
          q1=q1-1.0
      enddo 11
      v1(1)=n*(n+1)-m*(m+1)+c2/2.0           Initial guess for eigenvalue and function value.
      v2(2)=v1(1)
```

```
v2(1)=gamma*(1.-(v2(2)-c2)*dx/(2*(m+1)))
x1=-1.0+dx                              Set range of integration.
x2=1.0-dx
xf=0.                                   Fitting point.
call newt(v,NTOT,check)                 Find v that zeros function f in score.
if(check)then
    write(*,*)'shootf failed; bad initial guess'
else
    write(*,'(1x,t6,a)') 'mu(m,n)'
    write(*,'(1x,f12.6)') v1(1)
    goto 1
endif
END

SUBROUTINE load1(x1,v1,y)
INTEGER m,n
REAL c2,dx,gamma,x1,y1,v1(1),y(3)
COMMON /sphcom/ c2,gamma,dx,m,n
    Supplies starting values for integration at x = -1 + dx.
y(3)=v1(1)
if(mod(n-m,2).eq.0)then
    y1=gamma
else
    y1=-gamma
endif
y(2)=-(y(3)-c2)*y1/(2*(m+1))
y(1)=y1+y(2)*dx
return
END

SUBROUTINE load2(x2,v2,y)
INTEGER m,n
REAL c2,dx,gamma,x2,v2(2),y(3)
COMMON /sphcom/ c2,gamma,dx,m,n
    Supplies starting values for integration at x = 1 - dx.
y(3)=v2(2)
y(1)=v2(1)
y(2)=(y(3)-c2)*y(1)/(2*(m+1))
return
END

SUBROUTINE score(xf,y,f)
INTEGER i,m,n
REAL c2,gamma,dx,xf,f(3),y(3)
COMMON /sphcom/ c2,gamma,dx,m,n
    Tests whether solutions match at fitting point x = 0.
do 12 i=1,3
    f(i)=y(i)
enddo 12
return
END
```

CITED REFERENCES AND FURTHER READING:

Flammer, C. 1957, *Spheroidal Wave Functions* (Stanford, CA: Stanford University Press). [1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §21. [2]

Morse, P.M., and Feshbach, H. 1953, *Methods of Theoretical Physics*, Part II (New York: McGraw-Hill), pp. 1502ff. [3]

17.5 Automated Allocation of Mesh Points

In relaxation problems, you have to choose values for the independent variable at the mesh points. This is called *allocating* the grid or mesh. The usual procedure is to pick a plausible set of values and, if it works, to be content. If it doesn't work, increasing the number of points usually cures the problem.

If we know ahead of time where our solutions will be rapidly varying, we can put more grid points there and less elsewhere. Alternatively, we can solve the problem first on a uniform mesh and then examine the solution to see where we should add more points. We then repeat the solution with the improved grid. The object of the exercise is to allocate points in such a way as to represent the solution accurately.

It is also possible to automate the allocation of mesh points, so that it is done "dynamically" during the relaxation process. This powerful technique not only improves the accuracy of the relaxation method, but also (as we will see in the next section) allows internal singularities to be handled in quite a neat way. Here we learn how to accomplish the automatic allocation.

We want to focus attention on the independent variable x, and consider two alternative reparametrizations of it. The first, we term q; this is just the coordinate corresponding to the mesh points themselves, so that $q = 1$ at $\mathtt{k} = 1$, $q = 2$ at $\mathtt{k} = 2$, and so on. Between any two mesh points we have $\Delta q = 1$. In the change of independent variable in the ODEs from x to q,

$$\frac{d\mathbf{y}}{dx} = \mathbf{g} \tag{17.5.1}$$

becomes

$$\frac{d\mathbf{y}}{dq} = \mathbf{g}\frac{dx}{dq} \tag{17.5.2}$$

In terms of q, equation (17.5.2) as an FDE might be written

$$\mathbf{y}_k - \mathbf{y}_{k-1} - \tfrac{1}{2}\left[\left(\mathbf{g}\frac{dx}{dq}\right)_k + \left(\mathbf{g}\frac{dx}{dq}\right)_{k-1}\right] = 0 \tag{17.5.3}$$

or some related version. Note that dx/dq should accompany $\mathbf{g}$. The transformation between x and q depends only on the *Jacobian* dx/dq. Its reciprocal dq/dx is proportional to the density of mesh points.

Now, given the function $\mathbf{y}(x)$, or its approximation at the current stage of relaxation, we are supposed to have some idea of how we want to specify the density of mesh points. For example, we might want dq/dx to be larger where $\mathbf{y}$ is changing rapidly, or near to the boundaries, or both. In fact, we can probably make up a formula for what we would like dq/dx to be proportional to. The problem is that we do not know the proportionality constant. That is, the formula that we might invent would not have the correct integral over the whole range of x so as to make q vary from 1 to M, according to its definition. To solve this problem we introduce a second reparametrization $Q(q)$, where Q is a new independent variable. The relation between Q and q is taken to be *linear*, so that a mesh spacing formula for dQ/dx differs only in its unknown proportionality constant. A linear relation implies

$$\frac{d^2Q}{dq^2} = 0 \tag{17.5.4}$$

or, expressed in the usual manner as coupled first-order equations,

$$\frac{dQ(x)}{dq} = \psi \qquad \frac{d\psi}{dq} = 0 \tag{17.5.5}$$

where ψ is a new intermediate variable. We add these two equations to the set of ODEs being solved.

Completing the prescription, we add a third ODE that is just our desired mesh-density function, namely

$$\phi(x) = \frac{dQ}{dx} = \frac{dQ}{dq}\frac{dq}{dx} \tag{17.5.6}$$

17.5 Automated Allocation of Mesh Points

In relaxation problems, you have to choose values for the independent variable at the mesh points. This is called *allocating* the grid or mesh. The usual procedure is to pick a plausible set of values and, if it works, to be content. If it doesn't work, increasing the number of points usually cures the problem.

If we know ahead of time where our solutions will be rapidly varying, we can put more grid points there and less elsewhere. Alternatively, we can solve the problem first on a uniform mesh and then examine the solution to see where we should add more points. We then repeat the solution with the improved grid. The object of the exercise is to allocate points in such a way as to represent the solution accurately.

It is also possible to automate the allocation of mesh points, so that it is done "dynamically" during the relaxation process. This powerful technique not only improves the accuracy of the relaxation method, but also (as we will see in the next section) allows internal singularities to be handled in quite a neat way. Here we learn how to accomplish the automatic allocation.

We want to focus attention on the independent variable x, and consider two alternative reparametrizations of it. The first, we term q; this is just the coordinate corresponding to the mesh points themselves, so that $q = 1$ at $\mathtt{k} = 1$, $q = 2$ at $\mathtt{k} = 2$, and so on. Between any two mesh points we have $\Delta q = 1$. In the change of independent variable in the ODEs from x to q,

$$\frac{d\mathbf{y}}{dx} = \mathbf{g} \tag{17.5.1}$$

becomes

$$\frac{d\mathbf{y}}{dq} = \mathbf{g}\frac{dx}{dq} \tag{17.5.2}$$

In terms of q, equation (17.5.2) as an FDE might be written

$$\mathbf{y}_k - \mathbf{y}_{k-1} - \tfrac{1}{2}\left[\left(\mathbf{g}\frac{dx}{dq}\right)_k + \left(\mathbf{g}\frac{dx}{dq}\right)_{k-1}\right] = 0 \tag{17.5.3}$$

or some related version. Note that dx/dq should accompany $\mathbf{g}$. The transformation between x and q depends only on the *Jacobian* dx/dq. Its reciprocal dq/dx is proportional to the density of mesh points.

Now, given the function $\mathbf{y}(x)$, or its approximation at the current stage of relaxation, we are supposed to have some idea of how we want to specify the density of mesh points. For example, we might want dq/dx to be larger where $\mathbf{y}$ is changing rapidly, or near to the boundaries, or both. In fact, we can probably make up a formula for what we would like dq/dx to be proportional to. The problem is that we do not know the proportionality constant. That is, the formula that we might invent would not have the correct integral over the whole range of x so as to make q vary from 1 to M, according to its definition. To solve this problem we introduce a second reparametrization $Q(q)$, where Q is a new independent variable. The relation between Q and q is taken to be *linear*, so that a mesh spacing formula for dQ/dx differs only in its unknown proportionality constant. A linear relation implies

$$\frac{d^2Q}{dq^2} = 0 \tag{17.5.4}$$

or, expressed in the usual manner as coupled first-order equations,

$$\frac{dQ(x)}{dq} = \psi \qquad \frac{d\psi}{dq} = 0 \tag{17.5.5}$$

where ψ is a new intermediate variable. We add these two equations to the set of ODEs being solved.

Completing the prescription, we add a third ODE that is just our desired mesh-density function, namely

$$\phi(x) = \frac{dQ}{dx} = \frac{dQ}{dq}\frac{dq}{dx} \tag{17.5.6}$$

where $\phi(x)$ is chosen by us. Written in terms of the mesh variable q, this equation is

$$\frac{dx}{dq} = \frac{\psi}{\phi(x)} \tag{17.5.7}$$

Notice that $\phi(x)$ should be chosen to be positive definite, so that the density of mesh points is everywhere positive. Otherwise (17.5.7) can have a zero in its denominator.

To use automated mesh spacing, you add the three ODEs (17.5.5) and (17.5.7) to your set of equations, i.e., to the array `y(j,k)`. Now x becomes a dependent variable! Q and ψ also become new dependent variables. Normally, evaluating ϕ requires little extra work since it will be composed from pieces of the g's that exist anyway. The automated procedure allows one to investigate quickly how the numerical results might be affected by various strategies for mesh spacing. (A special case occurs if the desired mesh spacing function Q can be found analytically, i.e., dQ/dx is directly integrable. Then, you need to add only two equations, those in 17.5.5, and two new variables x, ψ.)

As an example of a typical strategy for implementing this scheme, consider a system with one dependent variable $y(x)$. We could set

$$dQ = \frac{dx}{\Delta} + \frac{|d\ln y|}{\delta} \tag{17.5.8}$$

or

$$\phi(x) = \frac{dQ}{dx} = \frac{1}{\Delta} + \left|\frac{dy/dx}{y\delta}\right| \tag{17.5.9}$$

where Δ and δ are constants that we choose. The first term would give a uniform spacing in x if it alone were present. The second term forces more grid points to be used where y is changing rapidly. The constants act to make every logarithmic change in y of an amount δ about as "attractive" to a grid point as a change in x of amount Δ. You adjust the constants according to taste. Other strategies are possible, such as a logarithmic spacing in x, replacing dx in the first term with $d\ln x$.

CITED REFERENCES AND FURTHER READING:

Eggleton, P. P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364.

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

17.6 Handling Internal Boundary Conditions or Singular Points

Singularities can occur in the interiors of two point boundary value problems. Typically, there is a point x_s at which a derivative must be evaluated by an expression of the form

$$S(x_s) = \frac{N(x_s, \mathbf{y})}{D(x_s, \mathbf{y})} \tag{17.6.1}$$

where the denominator $D(x_s, \mathbf{y}) = 0$. In physical problems with finite answers, singular points usually come with their own cure: Where $D \to 0$, there the physical solution $\mathbf{y}$ must be such as to make $N \to 0$ simultaneously, in such a way that the ratio takes on a meaningful value. This constraint on the solution $\mathbf{y}$ is often called a *regularity condition*. The condition that $D(x_s, \mathbf{y})$ satisfy some special constraint at x_s is entirely analogous to an extra boundary condition, an algebraic relation among the dependent variables that must hold at a point.

We discussed a related situation earlier, in §17.2, when we described the "fitting point method" to handle the task of integrating equations with singular behavior at the boundaries. In those problems you are unable to integrate from one side of the domain to the other.

where $\phi(x)$ is chosen by us. Written in terms of the mesh variable q, this equation is

$$\frac{dx}{dq} = \frac{\psi}{\phi(x)} \tag{17.5.7}$$

Notice that $\phi(x)$ should be chosen to be positive definite, so that the density of mesh points is everywhere positive. Otherwise (17.5.7) can have a zero in its denominator.

To use automated mesh spacing, you add the three ODEs (17.5.5) and (17.5.7) to your set of equations, i.e., to the array `y(j,k)`. Now x becomes a dependent variable! Q and ψ also become new dependent variables. Normally, evaluating ϕ requires little extra work since it will be composed from pieces of the g's that exist anyway. The automated procedure allows one to investigate quickly how the numerical results might be affected by various strategies for mesh spacing. (A special case occurs if the desired mesh spacing function Q can be found analytically, i.e., dQ/dx is directly integrable. Then, you need to add only two equations, those in 17.5.5, and two new variables x, ψ.)

As an example of a typical strategy for implementing this scheme, consider a system with one dependent variable $y(x)$. We could set

$$dQ = \frac{dx}{\Delta} + \frac{|d\ln y|}{\delta} \tag{17.5.8}$$

or

$$\phi(x) = \frac{dQ}{dx} = \frac{1}{\Delta} + \left|\frac{dy/dx}{y\delta}\right| \tag{17.5.9}$$

where Δ and δ are constants that we choose. The first term would give a uniform spacing in x if it alone were present. The second term forces more grid points to be used where y is changing rapidly. The constants act to make every logarithmic change in y of an amount δ about as "attractive" to a grid point as a change in x of amount Δ. You adjust the constants according to taste. Other strategies are possible, such as a logarithmic spacing in x, replacing dx in the first term with $d\ln x$.

CITED REFERENCES AND FURTHER READING:

Eggleton, P. P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364.

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

17.6 Handling Internal Boundary Conditions or Singular Points

Singularities can occur in the interiors of two point boundary value problems. Typically, there is a point x_s at which a derivative must be evaluated by an expression of the form

$$S(x_s) = \frac{N(x_s, \mathbf{y})}{D(x_s, \mathbf{y})} \tag{17.6.1}$$

where the denominator $D(x_s, \mathbf{y}) = 0$. In physical problems with finite answers, singular points usually come with their own cure: Where $D \to 0$, there the physical solution $\mathbf{y}$ must be such as to make $N \to 0$ simultaneously, in such a way that the ratio takes on a meaningful value. This constraint on the solution $\mathbf{y}$ is often called a *regularity condition*. The condition that $D(x_s, \mathbf{y})$ satisfy some special constraint at x_s is entirely analogous to an extra boundary condition, an algebraic relation among the dependent variables that must hold at a point.

We discussed a related situation earlier, in §17.2, when we described the "fitting point method" to handle the task of integrating equations with singular behavior at the boundaries. In those problems you are unable to integrate from one side of the domain to the other.

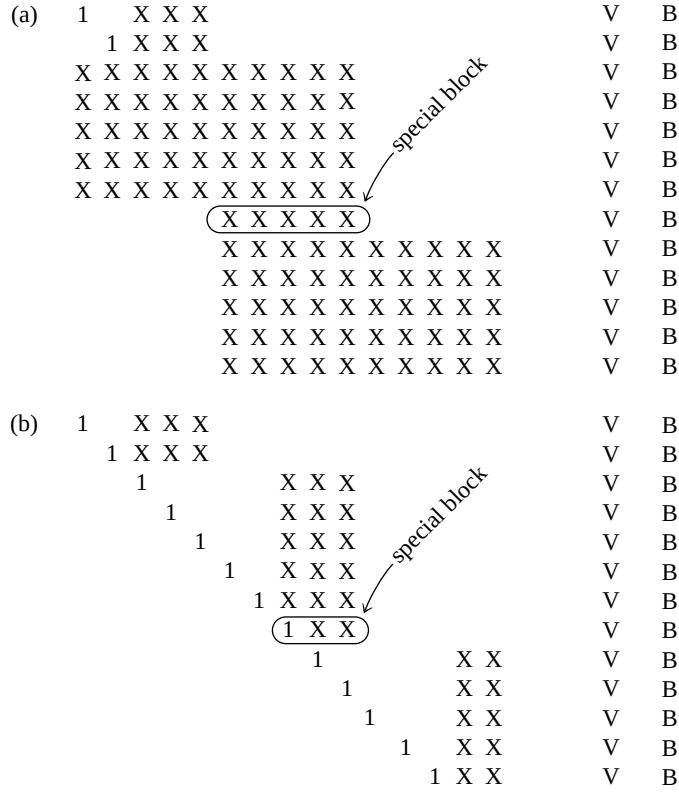


Figure 17.6.1. FDE matrix structure with an internal boundary condition. The internal condition introduces a special block. (a) Original form, compare with Figure 17.3.1; (b) final form, compare with Figure 17.3.2.

However, the ODEs do have well-behaved derivatives and solutions in the neighborhood of the singularity, so it is readily possible to integrate away from the point. Both the relaxation method and the method of "shooting" to a fitting point handle such problems easily. Also, in those problems the presence of singular behavior served to isolate some special boundary values that had to be satisfied to solve the equations.

The difference here is that we are concerned with singularities arising at intermediate points, where the location of the singular point depends on the solution, so is not known *a priori*. Consequently, we face a circular task: The singularity prevents us from finding a numerical solution, but we need a numerical solution to find its location. Such singularities are also associated with selecting a special value for some variable which allows the solution to satisfy the regularity condition at the singular point. Thus, internal singularities take on aspects of being internal boundary conditions.

One way of handling internal singularities is to treat the problem as a free boundary problem, as discussed at the end of §17.0. Suppose, as a simple example, we consider the equation

$$\frac{dy}{dx} = \frac{N(x,y)}{D(x,y)} \tag{17.6.2}$$

where N and D are required to pass through zero at some unknown point x_s. We add the equation

$$z \equiv x_s - x_1 \qquad \frac{dz}{dx} = 0 \tag{17.6.3}$$

where x_s is the unknown location of the singularity, and change the independent variable to t by setting

$$x - x_1 = tz, \qquad 0 \le t \le 1 \tag{17.6.4}$$

The boundary conditions at $t = 1$ become

$$N(x, y) = 0, \qquad D(x, y) = 0 \tag{17.6.5}$$

Use of an adaptive mesh as discussed in the previous section is another way to overcome the difficulties of an internal singularity. For the problem (17.6.2), we add the mesh spacing equations

$$\frac{dQ}{dq} = \psi \tag{17.6.6}$$

$$\frac{d\psi}{dq} = 0 \tag{17.6.7}$$

with a simple mesh spacing function that maps x uniformly into q, where q runs from 1 to M, the number of mesh points:

$$Q(x) = x - x_1, \quad \frac{dQ}{dx} = 1 \tag{17.6.8}$$

Having added three first-order differential equations, we must also add their corresponding boundary conditions. If there were no singularity, these could simply be

$$\text{at} \quad q = 1: \qquad x = x_1, \quad Q = 0 \tag{17.6.9}$$

$$\text{at} \quad q = M: \qquad x = x_2 \tag{17.6.10}$$

and a total of N values y_i specified at $q = 1$. In this case the problem is essentially an initial value problem with all boundary conditions specified at x_1 and the mesh spacing function is superfluous.

However, in the actual case at hand we impose the conditions

$$\text{at} \quad q = 1: \qquad x = x_1, \quad Q = 0 \tag{17.6.11}$$

$$\text{at} \quad q = M: \quad N(x, y) = 0, \quad D(x, y) = 0 \tag{17.6.12}$$

and $N - 1$ values y_i at $q = 1$. The "missing" y_i is to be adjusted, in other words, so as to make the solution go through the singular point in a regular (zero-over-zero) rather than irregular (finite-over-zero) manner. Notice also that these boundary conditions do not directly impose a value for x_2, which becomes an adjustable parameter that the code varies in an attempt to match the regularity condition.

In this example the singularity occurred at a boundary, and the complication arose because the location of the boundary was unknown. In other problems we might wish to continue the integration beyond the internal singularity. For the example given above, we could simply integrate the ODEs to the singular point, then as a separate problem recommence the integration from the singular point on as far we care to go. However, in other cases the singularity occurs internally, but does not completely determine the problem: There are still some more boundary conditions to be satisfied further along in the mesh. Such cases present no difficulty in principle, but do require some adaptation of the relaxation code given in §17.3. In effect all you need to do is to add a "special" block of equations at the mesh point where the internal boundary conditions occur, and do the proper bookkeeping.

Figure 17.6.1 illustrates a concrete example where the overall problem contains 5 equations with 2 boundary conditions at the first point, one "internal" boundary condition, and two final boundary conditions. The figure shows the structure of the overall matrix equations along the diagonal in the vicinity of the special block. In the middle of the domain, blocks typically involve 5 equations (rows) in 10 unknowns (columns). For each block prior to the special block, the initial boundary conditions provided enough information to zero the first two columns of the blocks. The five FDEs eliminate five more columns, and the final three columns need to be stored for the backsubstitution step (as described in §17.3). To handle the extra condition we break the normal cycle and add a special block with only one

equation: the internal boundary condition. This effectively reduces the required storage of unreduced coefficients by one column for the rest of the grid, and allows us to reduce to zero the first three columns of subsequent blocks. The subroutines `red, pinvs, bksub` can readily handle these cases with minor recoding, but each problem makes for a special case, and you will have to make the modifications as required.

CITED REFERENCES AND FURTHER READING:

London, R.A., and Flannery, B.P. 1982, *Astrophysical Journal*, vol. 258, pp. 260–269.

Chapter 18. Integral Equations and Inverse Theory

18.0 Introduction

Many people, otherwise numerically knowledgable, imagine that the numerical solution of integral equations must be an extremely arcane topic, since, until recently, it was almost never treated in numerical analysis textbooks. Actually there is a large and growing literature on the numerical solution of integral equations; several monographs have by now appeared [1-3]. One reason for the sheer volume of this activity is that there are many different kinds of equations, each with many different possible pitfalls; often many different algorithms have been proposed to deal with a single case.

There is a close correspondence between linear integral equations, which specify linear, integral relations among functions in an infinite-dimensional function space, and plain old linear equations, which specify analogous relations among vectors in a finite-dimensional vector space. Because this correspondence lies at the heart of most computational algorithms, it is worth making it explicit as we recall how integral equations are classified.

Fredholm equations involve definite integrals with fixed upper and lower limits. An *inhomogeneous Fredholm equation of the first kind* has the form

$$g(t) = \int_a^b K(t,s) f(s)\, ds \tag{18.0.1}$$

Here $f(t)$ is the unknown function to be solved for, while $g(t)$ is a known "right-hand side." (In integral equations, for some odd reason, the familiar "right-hand side" is conventionally written on the left!) The function of two variables, $K(t,s)$ is called the *kernel*. Equation (18.0.1) is analogous to the matrix equation

$$\mathbf{K} \cdot \mathbf{f} = \mathbf{g} \tag{18.0.2}$$

whose solution is $\mathbf{f} = \mathbf{K}^{-1} \cdot \mathbf{g}$, where $\mathbf{K}^{-1}$ is the matrix inverse. Like equation (18.0.2), equation (18.0.1) has a unique solution whenever g is nonzero (the homogeneous case with $g = 0$ is almost never useful) and K is invertible. However, as we shall see, this latter condition is as often the exception as the rule.

The analog of the finite-dimensional eigenvalue problem

$$(\mathbf{K} - \sigma \mathbf{1}) \cdot \mathbf{f} = \mathbf{g} \tag{18.0.3}$$

is called a *Fredholm equation of the second kind*, usually written

$$f(t) = \lambda \int_a^b K(t,s) f(s)\, ds + g(t) \tag{18.0.4}$$

Again, the notational conventions do not exactly correspond: λ in equation (18.0.4) is $1/\sigma$ in (18.0.3), while $\mathbf{g}$ is $-g/\lambda$. If g (or $\mathbf{g}$) is zero, then the equation is said to be *homogeneous*. If the kernel $K(t,s)$ is bounded, then, like equation (18.0.3), equation (18.0.4) has the property that its homogeneous form has solutions for at most a denumerably infinite set $\lambda = \lambda_n$, $n = 1, 2, \ldots$, the *eigenvalues*. The corresponding solutions $f_n(t)$ are the *eigenfunctions*. The eigenvalues are real if the kernel is symmetric.

In the *inhomogeneous* case of nonzero g (or $\mathbf{g}$), equations (18.0.3) and (18.0.4) are soluble *except* when λ (or σ) is an eigenvalue — because the integral operator (or matrix) is singular then. In integral equations this dichotomy is called *the Fredholm alternative*.

Fredholm equations of the first kind are often extremely ill-conditioned. Applying the kernel to a function is generally a smoothing operation, so the solution, which requires inverting the operator, will be extremely sensitive to small changes or errors in the input. Smoothing often actually loses information, and there is no way to get it back in an inverse operation. Specialized methods have been developed for such equations, which are often called *inverse problems*. In general, a method must augment the information given with some prior knowledge of the nature of the solution. This prior knowledge is then used, in one way or another, to restore lost information. We will introduce such techniques in §18.4.

Inhomogeneous Fredholm equations of the second kind are much less often ill-conditioned. Equation (18.0.4) can be rewritten as

$$\int_a^b [K(t,s) - \sigma\delta(t-s)] f(s)\, ds = -\sigma g(t) \tag{18.0.5}$$

where $\delta(t-s)$ is a Dirac delta function (and where we have changed from λ to its reciprocal σ for clarity). If σ is large enough in magnitude, then equation (18.0.5) is, in effect, diagonally dominant and thus well-conditioned. Only if σ is small do we go back to the ill-conditioned case.

Homogeneous Fredholm equations of the second kind are likewise not particularly ill-posed. If K is a smoothing operator, then it will map many f's to zero, or near-zero; there will thus be a large number of degenerate or nearly degenerate eigenvalues around $\sigma = 0$ ($\lambda \to \infty$), but this will cause no particular computational difficulties. In fact, we can now see that the magnitude of σ needed to rescue the inhomogeneous equation (18.0.5) from an ill-conditioned fate is generally much *less* than that required for diagonal dominance. Since the σ term shifts all eigenvalues, it is enough that it be large enough to shift a smoothing operator's forest of near-zero eigenvalues away from zero, so that the resulting operator becomes invertible (except, of course, at the discrete eigenvalues).

Volterra equations are a special case of Fredholm equations with $K(t,s) = 0$ for $s > t$. Chopping off the unnecessary part of the integration, Volterra equations are written in a form where the upper limit of integration is the independent variable t.

The *Volterra equation of the first kind*

$$g(t) = \int_a^t K(t,s)f(s)\,ds \tag{18.0.6}$$

has as its analog the matrix equation (now written out in components)

$$\sum_{j=1}^{k} K_{kj} f_j = g_k \tag{18.0.7}$$

Comparing with equation (18.0.2), we see that the Volterra equation corresponds to a matrix $\mathbf{K}$ that is lower (i.e., left) triangular, with zero entries above the diagonal. As we know from Chapter 2, such matrix equations are trivially soluble by forward substitution. Techniques for solving Volterra equations are similarly straightforward. When experimental measurement noise does not dominate, Volterra equations of the first kind tend *not* to be ill-conditioned; the upper limit to the integral introduces a sharp step that conveniently spoils any smoothing properties of the kernel.

The Volterra equation of the second kind is written

$$f(t) = \int_a^t K(t,s)f(s)\,ds + g(t) \tag{18.0.8}$$

whose matrix analog is the equation

$$(\mathbf{K} - \mathbf{1}) \cdot \mathbf{f} = \mathbf{g} \tag{18.0.9}$$

with $\mathbf{K}$ lower triangular. The reason there is no λ in these equations is that (i) in the inhomogeneous case (nonzero g) it can be absorbed into K, while (ii) in the homogeneous case ($g = 0$), it is a theorem that Volterra equations of the second kind with bounded kernels have no eigenvalues with square-integrable eigenfunctions.

We have specialized our definitions to the case of linear integral equations. The integrand in a nonlinear version of equation (18.0.1) or (18.0.6) would be $K(t, s, f(s))$ instead of $K(t,s)f(s)$; a nonlinear version of equation (18.0.4) or (18.0.8) would have an integrand $K(t, s, f(t), f(s))$. Nonlinear Fredholm equations are considerably more complicated than their linear counterparts. Fortunately, they do not occur as frequently in practice and we shall by and large ignore them in this chapter. By contrast, solving nonlinear Volterra equations usually involves only a slight modification of the algorithm for linear equations, as we shall see.

Almost all methods for solving integral equations numerically make use of *quadrature rules*, frequently Gaussian quadratures. This would be a good time for you to go back and review §4.5, especially the advanced material towards the end of that section.

In the sections that follow, we first discuss Fredholm equations of the second kind with smooth kernels (§18.1). Nontrivial quadrature rules come into the discussion, but we will be dealing with well-conditioned systems of equations. We then return to Volterra equations (§18.2), and find that simple and straightforward methods are generally satisfactory for these equations.

In §18.3 we discuss how to proceed in the case of singular kernels, focusing largely on Fredholm equations (both first and second kinds). Singularities require

special quadrature rules, but they are also sometimes blessings in disguise, since they can spoil a kernel's smoothing and make problems well-conditioned.

In §§18.4–18.7 we face up to the issues of inverse problems. §18.4 is an introduction to this large subject.

We should note here that wavelet transforms, already discussed in §13.10, are applicable not only to data compression and signal processing, but can also be used to transform some classes of integral equations into sparse linear problems that allow fast solution. You may wish to review §13.10 as part of reading this chapter.

Some subjects, such as *integro-differential equations*, we must simply declare to be beyond our scope. For a review of methods for integro-differential equations, see Brunner [4].

It should go without saying that this one short chapter can only barely touch on a few of the most basic methods involved in this complicated subject.

CITED REFERENCES AND FURTHER READING:

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [2]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.). [3]

Brunner, H. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical), pp. 18–38. [4]

Smithies, F. 1958, *Integral Equations* (Cambridge, U.K.: Cambridge University Press).

Kanwal, R.P. 1971, *Linear Integral Equations* (New York: Academic Press).

Green, C.D. 1969, *Integral Equation Methods* (New York: Barnes & Noble).

18.1 Fredholm Equations of the Second Kind

We desire a numerical solution for $f(t)$ in the equation

$$f(t) = \lambda \int_a^b K(t,s) f(s)\, ds + g(t) \tag{18.1.1}$$

The method we describe, a very basic one, is called the *Nystrom method*. It requires the choice of some approximate *quadrature rule*:

$$\int_a^b y(s)\, ds = \sum_{j=1}^{N} w_j y(s_j) \tag{18.1.2}$$

Here the set $\{w_j\}$ are the weights of the quadrature rule, while the N points $\{s_j\}$ are the abscissas.

What quadrature rule should we use? It is certainly possible to solve integral equations with low-order quadrature rules like the repeated trapezoidal or Simpson's

special quadrature rules, but they are also sometimes blessings in disguise, since they can spoil a kernel's smoothing and make problems well-conditioned.

In §§18.4–18.7 we face up to the issues of inverse problems. §18.4 is an introduction to this large subject.

We should note here that wavelet transforms, already discussed in §13.10, are applicable not only to data compression and signal processing, but can also be used to transform some classes of integral equations into sparse linear problems that allow fast solution. You may wish to review §13.10 as part of reading this chapter.

Some subjects, such as *integro-differential equations*, we must simply declare to be beyond our scope. For a review of methods for integro-differential equations, see Brunner [4].

It should go without saying that this one short chapter can only barely touch on a few of the most basic methods involved in this complicated subject.

CITED REFERENCES AND FURTHER READING:

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [2]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.). [3]

Brunner, H. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical), pp. 18–38. [4]

Smithies, F. 1958, *Integral Equations* (Cambridge, U.K.: Cambridge University Press).

Kanwal, R.P. 1971, *Linear Integral Equations* (New York: Academic Press).

Green, C.D. 1969, *Integral Equation Methods* (New York: Barnes & Noble).

18.1 Fredholm Equations of the Second Kind

We desire a numerical solution for $f(t)$ in the equation

$$f(t) = \lambda \int_a^b K(t,s) f(s)\, ds + g(t) \tag{18.1.1}$$

The method we describe, a very basic one, is called the *Nystrom method*. It requires the choice of some approximate *quadrature rule*:

$$\int_a^b y(s)\, ds = \sum_{j=1}^{N} w_j y(s_j) \tag{18.1.2}$$

Here the set $\{w_j\}$ are the weights of the quadrature rule, while the N points $\{s_j\}$ are the abscissas.

What quadrature rule should we use? It is certainly possible to solve integral equations with low-order quadrature rules like the repeated trapezoidal or Simpson's

rules. We will see, however, that the solution method involves $O(N^3)$ operations, and so the most efficient methods tend to use high-order quadrature rules to keep N as small as possible. For smooth, nonsingular problems, nothing beats Gaussian quadrature (e.g., Gauss-Legendre quadrature, §4.5). (For non-smooth or singular kernels, see §18.3.)

Delves and Mohamed [1] investigated methods more complicated than the Nystrom method. For straightforward Fredholm equations of the second kind, they concluded ". . . the clear winner of this contest has been the Nystrom routine . . . with the N-point Gauss-Legendre rule. This routine is extremely simple. . . . Such results are enough to make a numerical analyst weep."

If we apply the quadrature rule (18.1.2) to equation (18.1.1), we get

$$f(t) = \lambda \sum_{j=1}^{N} w_j K(t, s_j) f(s_j) + g(t) \tag{18.1.3}$$

Evaluate equation (18.1.3) at the quadrature points:

$$f(t_i) = \lambda \sum_{j=1}^{N} w_j K(t_i, s_j) f(s_j) + g(t_i) \tag{18.1.4}$$

Let f_i be the vector $f(t_i)$, g_i the vector $g(t_i)$, K_{ij} the matrix $K(t_i, s_j)$, and define

$$\widetilde{K}_{ij} = K_{ij} w_j \tag{18.1.5}$$

Then in matrix notation equation (18.1.4) becomes

$$(\mathbf{1} - \lambda \widetilde{\mathbf{K}}) \cdot \mathbf{f} = \mathbf{g} \tag{18.1.6}$$

This is a set of N linear algebraic equations in N unknowns that can be solved by standard triangular decomposition techniques (§2.3) — that is where the $O(N^3)$ operations count comes in. The solution is usually well-conditioned, unless λ is very close to an eigenvalue.

Having obtained the solution at the quadrature points $\{t_i\}$, how do you get the solution at some other point t? You do *not* simply use polynomial interpolation. This destroys all the accuracy you have worked so hard to achieve. Nystrom's key observation was that you should use equation (18.1.3) as an interpolatory formula, maintaining the accuracy of the solution.

We here give two subroutines for use with linear Fredholm equations of the second kind. The routine `fred2` sets up equation (18.1.6) and then solves it by LU decomposition with calls to the routines `ludcmp` and `lubksb`. The Gauss-Legendre quadrature is implemented by first getting the weights and abscissas with a call to `gauleg`. Routine `fred2` requires that you provide an external function that returns $g(t)$ and another that returns λK_{ij}. It then returns the solution f at the quadrature points. It also returns the quadrature points and weights. These are used by the second routine `fredin` to carry out the Nystrom interpolation of equation (18.1.3) and return the value of f at any point in the interval $[a, b]$.

```
      SUBROUTINE fred2(n,a,b,t,f,w,g,ak)
      INTEGER n,NMAX
      REAL a,b,f(n),t(n),w(n),g,ak
      EXTERNAL ak,g
      PARAMETER (NMAX=200)
C     USES ak,g,gauleg,lubksb,ludcmp
```

Solves a linear Fredholm equation of the second kind. On input, a and b are the limits of integration, and n is the number of points to use in the Gaussian quadrature. g and ak are user-supplied external functions that respectively return $g(t)$ and $\lambda K(t,s)$. The routine returns arrays t(1:n) and f(1:n) containing the abscissas t_i of the Gaussian quadrature and the solution f at these abscissas. Also returned is the array w(1:n) of Gaussian weights for use with the Nystrom interpolation routine fredin.

```
      INTEGER i,j,indx(NMAX)
      REAL d,omk(NMAX,NMAX)
      if(n.gt.NMAX) pause 'increase NMAX in fred2'
      call gauleg(a,b,t,w,n)              Replace gauleg with another routine if not using
      do 12 i=1,n                             Gauss-Legendre quadrature.
        do 11 j=1,n                       Form 1 − λK.
          if(i.eq.j)then
            omk(i,j)=1.
          else
            omk(i,j)=0.
          endif
          omk(i,j)=omk(i,j)-ak(t(i),t(j))*w(j)
        enddo 11
        f(i)=g(t(i))
      enddo 12
      call ludcmp(omk,n,NMAX,indx,d)      Solve linear equations.
      call lubksb(omk,n,NMAX,indx,f)
      return
      END


      FUNCTION fredin(x,n,a,b,t,f,w,g,ak)
      INTEGER n
      REAL fredin,a,b,x,f(n),t(n),w(n),g,ak
      EXTERNAL ak,g
C     USES ak,g
```

Given arrays t(1:n) and w(1:n) containing the abscissas and weights of the Gaussian quadrature, and given the solution array f(1:n) from fred2, this function returns the value of f at x using the Nystrom interpolation formula. On input, a and b are the limits of integration, and n is the number of points used in the Gaussian quadrature. g and ak are user-supplied external functions that respectively return $g(t)$ and $\lambda K(t,s)$.

```
      INTEGER i
      REAL sum
      sum=0.
      do 11 i=1,n
        sum=sum+ak(x,t(i))*w(i)*f(i)
      enddo 11
      fredin=g(x)+sum
      return
      END
```

One disadvantage of a method based on Gaussian quadrature is that there is no simple way to obtain an estimate of the error in the result. The best practical method is to increase N by 50%, say, and treat the difference between the two estimates as a conservative estimate of the error in the result obtained with the larger value of N.

Turn now to solutions of the homogeneous equation. If we set $\lambda = 1/\sigma$ and $\mathbf{g} = 0$, then equation (18.1.6) becomes a standard eigenvalue equation

$$\widetilde{\mathbf{K}} \cdot \mathbf{f} = \sigma \mathbf{f} \tag{18.1.7}$$

which we can solve with any convenient matrix eigenvalue routine (see Chapter 11). Note that if our original problem had a symmetric kernel, then the matrix $\mathbf{K}$ is symmetric. However, since the weights w_j are not equal for most quadrature rules, the matrix $\widetilde{\mathbf{K}}$ (equation 18.1.5) is not symmetric. The matrix eigenvalue problem is much easier for symmetric matrices, and so we should restore the symmetry if possible. Provided the weights are positive (which they are for Gaussian quadrature), we can define the diagonal matrix $\mathbf{D} = \text{diag}(w_j)$ and its square root, $\mathbf{D}^{1/2} = \text{diag}(\sqrt{w_j})$. Then equation (18.1.7) becomes

$$\mathbf{K} \cdot \mathbf{D} \cdot \mathbf{f} = \sigma \mathbf{f}$$

Multiplying by $\mathbf{D}^{1/2}$, we get

$$\left(\mathbf{D}^{1/2} \cdot \mathbf{K} \cdot \mathbf{D}^{1/2}\right) \cdot \mathbf{h} = \sigma \mathbf{h} \tag{18.1.8}$$

where $\mathbf{h} = \mathbf{D}^{1/2} \cdot \mathbf{f}$. Equation (18.1.8) is now in the form of a symmetric eigenvalue problem.

Solution of equations (18.1.7) or (18.1.8) will in general give N eigenvalues, where N is the number of quadrature points used. For square-integrable kernels, these will provide good approximations to the lowest N eigenvalues of the integral equation. Kernels of *finite rank* (also called *degenerate* or *separable* kernels) have only a finite number of nonzero eigenvalues (possibly none). You can diagnose this situation by a cluster of eigenvalues σ that are zero to machine precision. The number of nonzero eigenvalues will stay constant as you increase N to improve their accuracy. Some care is required here: A nondegenerate kernel can have an infinite number of eigenvalues that have an accumulation point at $\sigma = 0$. You distinguish the two cases by the behavior of the solution as you increase N. If you suspect a degenerate kernel, you will usually be able to solve the problem by analytic techniques described in all the textbooks.

CITED REFERENCES AND FURTHER READING:

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.).

18.2 Volterra Equations

Let us now turn to Volterra equations, of which our prototype is the Volterra equation of the second kind,

$$f(t) = \int_a^t K(t,s)f(s)\,ds + g(t) \tag{18.2.1}$$

Most algorithms for Volterra equations march out from $t = a$, building up the solution as they go. In this sense they resemble not only forward substitution (as discussed in §18.0), but also initial-value problems for ordinary differential equations. In fact, many algorithms for ODEs have counterparts for Volterra equations.

The simplest way to proceed is to solve the equation on a mesh with uniform spacing:

$$t_i = a + ih, \quad i = 0, 1, \ldots, N, \qquad h \equiv \frac{b-a}{N} \tag{18.2.2}$$

To do so, we must choose a quadrature rule. For a uniform mesh, the simplest scheme is the trapezoidal rule, equation (4.1.11):

$$\int_a^{t_i} K(t_i,s)f(s)\,ds = h\left(\tfrac{1}{2}K_{i0}f_0 + \sum_{j=1}^{i-1} K_{ij}f_j + \tfrac{1}{2}K_{ii}f_i\right) \tag{18.2.3}$$

Thus the trapezoidal method for equation (18.2.1) is:

$$\begin{aligned} f_0 &= g_0 \\ (1 - \tfrac{1}{2}hK_{ii})f_i &= h\left(\tfrac{1}{2}K_{i0}f_0 + \sum_{j=1}^{i-1} K_{ij}f_j\right) + g_i, \qquad i = 1, \ldots, N \end{aligned} \tag{18.2.4}$$

(For a Volterra equation of the first kind, the leading 1 on the left would be absent, and g would have opposite sign, with corresponding straightforward changes in the rest of the discussion.)

Equation (18.2.4) is an explicit prescription that gives the solution in $O(N^2)$ operations. Unlike Fredholm equations, it is not necessary to solve a system of linear equations. Volterra equations thus usually involve less work than the corresponding Fredholm equations which, as we have seen, do involve the inversion of, sometimes large, linear systems.

The efficiency of solving Volterra equations is somewhat counterbalanced by the fact that *systems* of these equations occur more frequently in practice. If we interpret equation (18.2.1) as a *vector* equation for the vector of m functions $f(t)$, then the kernel $K(t,s)$ is an $m \times m$ matrix. Equation (18.2.4) must now also be understood as a vector equation. For each i, we have to solve the $m \times m$ set of linear algebraic equations by Gaussian elimination.

The routine `voltra` below implements this algorithm. You must supply an external function that returns the kth function of the vector $g(t)$ at the point t, and another that returns the (k,l) element of the matrix $K(t,s)$ at (t,s). The routine `voltra` then returns the vector $f(t)$ at the regularly spaced points t_i.

18.2 Volterra Equations

Let us now turn to Volterra equations, of which our prototype is the Volterra equation of the second kind,

$$f(t) = \int_a^t K(t,s)f(s)\,ds + g(t) \tag{18.2.1}$$

Most algorithms for Volterra equations march out from $t = a$, building up the solution as they go. In this sense they resemble not only forward substitution (as discussed in §18.0), but also initial-value problems for ordinary differential equations. In fact, many algorithms for ODEs have counterparts for Volterra equations.

The simplest way to proceed is to solve the equation on a mesh with uniform spacing:

$$t_i = a + ih, \quad i = 0, 1, \ldots, N, \qquad h \equiv \frac{b-a}{N} \tag{18.2.2}$$

To do so, we must choose a quadrature rule. For a uniform mesh, the simplest scheme is the trapezoidal rule, equation (4.1.11):

$$\int_a^{t_i} K(t_i,s)f(s)\,ds = h\left(\tfrac{1}{2}K_{i0}f_0 + \sum_{j=1}^{i-1} K_{ij}f_j + \tfrac{1}{2}K_{ii}f_i\right) \tag{18.2.3}$$

Thus the trapezoidal method for equation (18.2.1) is:

$$\begin{aligned} f_0 &= g_0 \\ (1 - \tfrac{1}{2}hK_{ii})f_i &= h\left(\tfrac{1}{2}K_{i0}f_0 + \sum_{j=1}^{i-1} K_{ij}f_j\right) + g_i, \qquad i = 1, \ldots, N \end{aligned} \tag{18.2.4}$$

(For a Volterra equation of the first kind, the leading 1 on the left would be absent, and g would have opposite sign, with corresponding straightforward changes in the rest of the discussion.)

Equation (18.2.4) is an explicit prescription that gives the solution in $O(N^2)$ operations. Unlike Fredholm equations, it is not necessary to solve a system of linear equations. Volterra equations thus usually involve less work than the corresponding Fredholm equations which, as we have seen, do involve the inversion of, sometimes large, linear systems.

The efficiency of solving Volterra equations is somewhat counterbalanced by the fact that *systems* of these equations occur more frequently in practice. If we interpret equation (18.2.1) as a *vector* equation for the vector of m functions $f(t)$, then the kernel $K(t,s)$ is an $m \times m$ matrix. Equation (18.2.4) must now also be understood as a vector equation. For each i, we have to solve the $m \times m$ set of linear algebraic equations by Gaussian elimination.

The routine `voltra` below implements this algorithm. You must supply an external function that returns the kth function of the vector $g(t)$ at the point t, and another that returns the (k,l) element of the matrix $K(t,s)$ at (t,s). The routine `voltra` then returns the vector $f(t)$ at the regularly spaced points t_i.

```
      SUBROUTINE voltra(n,m,t0,h,t,f,g,ak)
      INTEGER m,n,MMAX
      REAL h,t0,f(m,n),t(n),g,ak
      EXTERNAL ak,g
      PARAMETER (MMAX=5)
C     USES ak,g,lubksb,ludcmp
```

Solves a set of m linear Volterra equations of the second kind using the extended trapezoidal rule. On input, t0 is the starting point of the integration and n-1 is the number of steps of size h to be taken. g(k,t) is a user-supplied external function that returns $g_k(t)$, while ak(k,l,t,s) is another user-supplied external function that returns the (k,l) element of the matrix $K(t,s)$. The solution is returned in f(1:m,1:n), with the corresponding abscissas in t(1:n).

```
      INTEGER i,j,k,l,indx(MMAX)
      REAL d,sum,a(MMAX,MMAX),b(MMAX)
      t(1)=t0
      do 11 k=1,m                              Initialize.
        f(k,1)=g(k,t(1))
      enddo 11
      do 16 i=2,n                              Take a step h.
        t(i)=t(i-1)+h
        do 14 k=1,m
          sum=g(k,t(i))                        Accumulate right-hand side of linear equations in
          do 13 l=1,m                              sum.
            sum=sum+0.5*h*ak(k,l,t(i),t(1))*f(l,1)
            do 12 j=2,i-1
              sum=sum+h*ak(k,l,t(i),t(j))*f(l,j)
            enddo 12
            if(k.eq.l)then                     Left-hand side goes in matrix a.
              a(k,l)=1.
            else
              a(k,l)=0.
            endif
            a(k,l)=a(k,l)-0.5*h*ak(k,l,t(i),t(i))
          enddo 13
          b(k)=sum
        enddo 14
        call ludcmp(a,m,MMAX,indx,d)           Solve linear equations.
        call lubksb(a,m,MMAX,indx,b)
        do 15 k=1,m
          f(k,i)=b(k)
        enddo 15
      enddo 16
      return
      END
```

For nonlinear Volterra equations, equation (18.2.4) holds with the product $K_{ii}f_i$ replaced by $K_{ii}(f_i)$, and similarly for the other two products of K's and f's. Thus for each i we solve a nonlinear equation for f_i with a known right-hand side. Newton's method (§9.4 or §9.6) with an initial guess of f_{i-1} usually works very well provided the stepsize is not too big.

Higher-order methods for solving Volterra equations are, in our opinion, not as important as for Fredholm equations, since Volterra equations are relatively easy to solve. However, there is an extensive literature on the subject. Several difficulties arise. First, any method that achieves higher order by operating on several quadrature points simultaneously will need a special method to get started, when values at the first few points are not yet known.

Second, stable quadrature rules can give rise to unexpected instabilities in integral equations. For example, suppose we try to replace the trapezoidal rule in

the algorithm above with Simpson's rule. Simpson's rule naturally integrates over an interval $2h$, so we easily get the function values at the even mesh points. For the odd mesh points, we could try appending one panel of trapezoidal rule. But to which end of the integration should we append it? We could do one step of trapezoidal rule followed by all Simpson's rule, or Simpson's rule with one step of trapezoidal rule at the end. Surprisingly, the former scheme is unstable, while the latter is fine!

A simple approach that can be used with the trapezoidal method given above is Richardson extrapolation: Compute the solution with stepsize h and $h/2$. Then, assuming the error scales with h^2, compute

$$f_{\rm E} = \frac{4f(h/2) - f(h)}{3} \tag{18.2.5}$$

This procedure can be repeated as with Romberg integration.

The general consensus is that the best of the higher order methods is the *block-by-block method* (see [1]). Another important topic is the use of variable stepsize methods, which are much more efficient if there are sharp features in K or f. Variable stepsize methods are quite a bit more complicated than their counterparts for differential equations; we refer you to the literature [1,2] for a discussion.

You should also be on the lookout for singularities in the integrand. If you find them, then look to §18.3 for additional ideas.

CITED REFERENCES AND FURTHER READING:

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [1]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [2]

18.3 Integral Equations with Singular Kernels

Many integral equations have singularities in either the kernel or the solution or both. A simple quadrature method will show poor convergence with N if such singularities are ignored. There is sometimes art in how singularities are best handled.

We start with a few straightforward suggestions:

1. Integrable singularities can often be removed by a change of variable. For example, the singular behavior $K(t,s) \sim s^{1/2}$ or $s^{-1/2}$ near $s = 0$ can be removed by the transformation $z = s^{1/2}$. Note that we are assuming that the singular behavior is confined to K, whereas the quadrature actually involves the product $K(t,s)f(s)$, and it is this product that must be "fixed." Ideally, you must deduce the singular nature of the product before you try a numerical solution, and take the appropriate action. Commonly, however, a singular kernel does *not* produce a singular solution $f(t)$. (The highly singular kernel $K(t,s) = \delta(t-s)$ is simply the identity operator, for example.)

2. If $K(t,s)$ can be factored as $w(s)\overline{K}(t,s)$, where $w(s)$ is singular and $\overline{K}(t,s)$ is smooth, then a Gaussian quadrature based on $w(s)$ as a weight function will work well. Even if the factorization is only approximate, the convergence is often improved dramatically. All you have to do is replace `gauleg` in the routine `fred2` by another quadrature routine. Section 4.5 explained how to construct such quadratures; or you can find tabulated abscissas and weights in the standard references [1,2]. You must of course supply $\overline{K}$ instead of K.

the algorithm above with Simpson's rule. Simpson's rule naturally integrates over an interval $2h$, so we easily get the function values at the even mesh points. For the odd mesh points, we could try appending one panel of trapezoidal rule. But to which end of the integration should we append it? We could do one step of trapezoidal rule followed by all Simpson's rule, or Simpson's rule with one step of trapezoidal rule at the end. Surprisingly, the former scheme is unstable, while the latter is fine!

A simple approach that can be used with the trapezoidal method given above is Richardson extrapolation: Compute the solution with stepsize h and $h/2$. Then, assuming the error scales with h^2, compute

$$f_{\rm E} = \frac{4f(h/2) - f(h)}{3} \tag{18.2.5}$$

This procedure can be repeated as with Romberg integration.

The general consensus is that the best of the higher order methods is the *block-by-block method* (see [1]). Another important topic is the use of variable stepsize methods, which are much more efficient if there are sharp features in K or f. Variable stepsize methods are quite a bit more complicated than their counterparts for differential equations; we refer you to the literature [1,2] for a discussion.

You should also be on the lookout for singularities in the integrand. If you find them, then look to §18.3 for additional ideas.

CITED REFERENCES AND FURTHER READING:

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [1]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [2]

18.3 Integral Equations with Singular Kernels

Many integral equations have singularities in either the kernel or the solution or both. A simple quadrature method will show poor convergence with N if such singularities are ignored. There is sometimes art in how singularities are best handled.

We start with a few straightforward suggestions:

1. Integrable singularities can often be removed by a change of variable. For example, the singular behavior $K(t,s) \sim s^{1/2}$ or $s^{-1/2}$ near $s = 0$ can be removed by the transformation $z = s^{1/2}$. Note that we are assuming that the singular behavior is confined to K, whereas the quadrature actually involves the product $K(t,s)f(s)$, and it is this product that must be "fixed." Ideally, you must deduce the singular nature of the product before you try a numerical solution, and take the appropriate action. Commonly, however, a singular kernel does *not* produce a singular solution $f(t)$. (The highly singular kernel $K(t,s) = \delta(t-s)$ is simply the identity operator, for example.)

2. If $K(t,s)$ can be factored as $w(s)\overline{K}(t,s)$, where $w(s)$ is singular and $\overline{K}(t,s)$ is smooth, then a Gaussian quadrature based on $w(s)$ as a weight function will work well. Even if the factorization is only approximate, the convergence is often improved dramatically. All you have to do is replace `gauleg` in the routine `fred2` by another quadrature routine. Section 4.5 explained how to construct such quadratures; or you can find tabulated abscissas and weights in the standard references [1,2]. You must of course supply $\overline{K}$ instead of K.

This method is a special case of the *product Nystrom method* [3,4], where one factors out a singular term $p(t,s)$ depending on both t and s from K and constructs suitable weights for its Gaussian quadrature. The calculations in the general case are quite cumbersome, because the weights depend on the chosen $\{t_i\}$ as well as the form of $p(t,s)$.

We prefer to implement the product Nystrom method on a uniform grid, with a quadrature scheme that generalizes the extended Simpson's 3/8 rule (equation 4.1.5) to arbitrary weight functions. We discuss this in the subsections below.

3. Special quadrature formulas are also useful when the kernel is not strictly singular, but is "almost" so. One example is when the kernel is concentrated near $t = s$ on a scale much smaller than the scale on which the solution $f(t)$ varies. In that case, a quadrature formula can be based on locally approximating $f(s)$ by a polynomial or spline, while calculating the first few *moments* of the kernel $K(t,s)$ at the tabulation points t_i. In such a scheme the narrow width of the kernel becomes an asset, rather than a liability: The quadrature becomes exact as the width of the kernel goes to zero.

4. An infinite range of integration is also a form of singularity. Truncating the range at a large finite value should be used only as a last resort. If the kernel goes rapidly to zero, then a Gauss-Laguerre [$w \sim \exp(-\alpha s)$] or Gauss-Hermite [$w \sim \exp(-s^2)$] quadrature should work well. Long-tailed functions often succumb to the transformation

$$s = \frac{2\alpha}{z+1} - \alpha \tag{18.3.1}$$

which maps $0 < s < \infty$ to $1 > z > -1$ so that Gauss-Legendre integration can be used. Here $\alpha > 0$ is a constant that you adjust to improve the convergence.

5. A common situation in practice is that $K(t,s)$ is singular along the diagonal line $t = s$. Here the Nystrom method fails completely because the kernel gets evaluated at (t_i, s_i). *Subtraction of the singularity* is one possible cure:

$$\begin{aligned}\int_a^b K(t,s)f(s)\,ds &= \int_a^b K(t,s)[f(s)-f(t)]\,ds + \int_a^b K(t,s)f(t)\,ds \\ &= \int_a^b K(t,s)[f(s)-f(t)]\,ds + r(t)f(t)\end{aligned} \tag{18.3.2}$$

where $r(t) = \int_a^b K(t,s)\,ds$ is computed analytically or numerically. If the first term on the right-hand side is now regular, we can use the Nystrom method. Instead of equation (18.1.4), we get

$$f_i = \lambda \sum_{\substack{j=1 \\ j\neq i}}^{N} w_j K_{ij}[f_j - f_i] + \lambda r_i f_i + g_i \tag{18.3.3}$$

Sometimes the subtraction process must be repeated before the kernel is completely regularized. See [3] for details. (And read on for a different, we think better, way to handle diagonal singularities.)

Quadrature on a Uniform Mesh with Arbitrary Weight

It is possible in general to find n-point linear quadrature rules that approximate the integral of a function $f(x)$, times an arbitrary weight function $w(x)$, over an arbitrary range of integration (a,b), as the sum of weights times n evenly spaced values of the function $f(x)$, say at $x = kh, (k+1)h, \ldots, (k+n-1)h$. The general scheme for deriving such quadrature rules is to write down the n linear equations that must be satisfied if the quadrature rule is to be exact for the n functions $f(x) = \text{const}, x, x^2, \ldots, x^{n-1}$, and then solve these for the coefficients. This can be done analytically, once and for all, if the moments of the weight function over the same range of integration,

$$W_n \equiv \frac{1}{h^n}\int_a^b x^n w(x)dx \tag{18.3.4}$$

are assumed to be known. Here the prefactor h^{-n} is chosen to make W_n scale as h if (as in the usual case) $b-a$ is proportional to h.

Carrying out this prescription for the four-point case gives the result

$$\begin{aligned}\int_a^b w(x)f(x)dx = \\ &\frac{1}{6}f(kh)\Big[(k+1)(k+2)(k+3)W_0 - (3k^2+12k+11)W_1 + 3(k+2)W_2 - W_3\Big] \\ &+\frac{1}{2}f([k+1]h)\Big[-k(k+2)(k+3)W_0 + (3k^2+10k+6)W_1 - (3k+5)W_2 + W_3\Big] \\ &+\frac{1}{2}f([k+2]h)\Big[k(k+1)(k+3)W_0 - (3k^2+8k+3)W_1 + (3k+4)W_2 - W_3\Big] \\ &+\frac{1}{6}f([k+3]h)\Big[-k(k+1)(k+2)W_0 + (3k^2+6k+2)W_1 - 3(k+1)W_2 + W_3\Big]\end{aligned} \tag{18.3.5}$$

While the terms in brackets superficially appear to scale as k^2, there is typically cancellation at both $O(k^2)$ and $O(k)$.

Equation (18.3.5) can be specialized to various choices of (a,b). The obvious choice is $a=kh$, $b=(k+3)h$, in which case we get a four-point quadrature rule that generalizes Simpson's 3/8 rule (equation 4.1.5). In fact, we can recover this special case by setting $w(x)=1$, in which case (18.3.4) becomes

$$W_n = \frac{h}{n+1}[(k+3)^{n+1} - k^{n+1}] \tag{18.3.6}$$

The four terms in square brackets equation (18.3.5) each become independent of k, and (18.3.5) in fact reduces to

$$\int_{kh}^{(k+3)h} f(x)dx = \frac{3h}{8}f(kh) + \frac{9h}{8}f([k+1]h) + \frac{9h}{8}f([k+2]h) + \frac{3h}{8}f([k+3]h) \tag{18.3.7}$$

Back to the case of general $w(x)$, some other choices for a and b are also useful. For example, we may want to choose (a,b) to be $([k+1]h,[k+3]h)$ or $([k+2]h,[k+3]h)$, allowing us to finish off an extended rule whose number of intervals is not a multiple of three, without loss of accuracy: The integral will be estimated using the four values $f(kh),\ldots,f([k+3]h)$. Even more useful is to choose (a,b) to be $([k+1]h,[k+2]h)$, thus using four points to integrate a centered single interval. These weights, when sewed together into an extended formula, give quadrature schemes that have smooth coefficients, i.e., without the Simpson-like $2,4,2,4,2$ alternation. (In fact, this was the technique that we used to derive equation 4.1.14, which you may now wish to reexamine.)

All these rules are of the same order as the extended Simpson's rule, that is, exact for $f(x)$ a cubic polynomial. Rules of lower order, if desired, are similarly obtained. The three point formula is

$$\begin{aligned}\int_a^b w(x)f(x)dx = &\frac{1}{2}f(kh)\Big[(k+1)(k+2)W_0 - (2k+3)W_1 + W_2\Big] \\ &+f([k+1]h)\Big[-k(k+2)W_0 + 2(k+1)W_1 - W_2\Big] \\ &+\frac{1}{2}f([k+2]h)\Big[k(k+1)W_0 - (2k+1)W_1 + W_2\Big]\end{aligned} \tag{18.3.8}$$

Here the simple special case is to take, $w(x)=1$, so that

$$W_n = \frac{h}{n+1}[(k+2)^{n+1} - k^{n+1}] \tag{18.3.9}$$

Then equation (18.3.8) becomes Simpson's rule,

$$\int_{kh}^{(k+2)h} f(x)dx = \frac{h}{3}f(kh) + \frac{4h}{3}f([k+1]h) + \frac{h}{3}f([k+2]h) \tag{18.3.10}$$

For nonconstant weight functions $w(x)$, however, equation (18.3.8) gives rules of one order less than Simpson, since they do not benefit from the extra symmetry of the constant case.

The two point formula is simply

$$\int_{kh}^{(k+1)h} w(x)f(x)dx = f(kh)[(k+1)W_0 - W_1] + f([k+1]h)[-kW_0 + W_1] \quad (18.3.11)$$

Here is a routine `wwghts` that uses the above formulas to return an extended N-point quadrature rule for the interval $(a, b) = (0, [N-1]h)$. Input to `wwghts` is a user-supplied routine, `kermom`, that is called to get the first four *indefinite-integral* moments of $w(x)$, namely

$$F_m(y) \equiv \int^y s^m w(s)ds \qquad m = 0, 1, 2, 3 \quad (18.3.12)$$

(The lower limit is arbitrary and can be chosen for convenience.) Cautionary note: When called with $N < 4$, `wwghts` returns a rule of lower order than Simpson; you should structure your problem to avoid this.

```
      SUBROUTINE wwghts(wghts,n,h,kermom)
      INTEGER n
      REAL wghts(n),h
      EXTERNAL kermom
C     USES kermom
        Constructs in wghts(1:n) weights for the n-point equal-interval quadrature from 0 to
        (n−1)h of a function f(x) times an arbitrary (possibly singular) weight function w(x) whose
        indefinite-integral moments F_n(y) are provided by the user-supplied subroutine kermom.
      INTEGER j,k
      DOUBLE PRECISION wold(4),wnew(4),w(4),hh,hi,c,fac,a,b
      hh=h                                  Double precision on internal calculations even though
      hi=1.d0/hh                                the interface is in single precision.
      do 11 j=1,n                           Zero all the weights so we can sum into them.
        wghts(j)=0.
      enddo 11
      call kermom(wold,0.d0,4)              Evaluate indefinite integrals at lower end.
      if (n.ge.4) then                      Use highest available order.
        b=0.d0                              For another problem, you might change this lower
        do 14 j=1,n-3                             limit.
          c=j-1                             This is called k in equation (18.3.5).
          a=b                               Set upper and lower limits for this step.
          b=a+hh
          if (j.eq.n-3) b=(n-1)*hh          Last interval: go all the way to end.
          call kermom(wnew,b,4)
          fac=1.d0
          do 12 k=1,4                       Equation (18.3.4).
            w(k)=(wnew(k)-wold(k))*fac
            fac=fac*hi
          enddo 12
          wghts(j)=wghts(j)+                Equation (18.3.5).
     *         ((c+1.d0)*(c+2.d0)*(c+3.d0)*w(1)
     *         -(11.d0+c*(12.d0+c*3.d0))*w(2)
     *         +3.d0*(c+2.d0)*w(3)-w(4))/6.d0
          wghts(j+1)=wghts(j+1)+
     *         (-c*(c+2.d0)*(c+3.d0)*w(1)
     *         +(6.d0+c*(10.d0+c*3.d0))*w(2)
     *         -(3.d0*c+5.d0)*w(3)+w(4))*.5d0
          wghts(j+2)=wghts(j+2)+
     *         (c*(c+1.d0)*(c+3.d0)*w(1)
     *         -(3.d0+c*(8.d0+c*3.d0))*w(2)
     *         +(3.d0*c+4.d0)*w(3)-w(4))*.5d0
          wghts(j+3)=wghts(j+3)+
     *         (-c*(c+1.d0)*(c+2.d0)*w(1)
     *         +(2.d0+c*(6.d0+c*3.d0))*w(2)
     *         -3.d0*(c+1.d0)*w(3)+w(4))/6.d0
          do 13 k=1,4                       Reset lower limits for moments.
```

```
            wold(k)=wnew(k)
          enddo 13
      enddo 14
else if (n.eq.3) then                Lower-order cases; not recommended.
      call kermom(wnew,hh+hh,3)
      w(1)=wnew(1)-wold(1)
      w(2)=hi*(wnew(2)-wold(2))
      w(3)=hi**2*(wnew(3)-wold(3))
      wghts(1)=w(1)-1.5d0*w(2)+0.5d0*w(3)
      wghts(2)=2.d0*w(2)-w(3)
      wghts(3)=0.5d0*(w(3)-w(2))
else if (n.eq.2) then
      call kermom(wnew,hh,2)
      wghts(2)=hi*(wnew(2)-wold(2))
      wghts(1)=wnew(1)-wold(1)-wghts(2)
endif
END
```

We will now give an example of how to apply `wwghts` to a singular integral equation.

Worked Example: A Diagonally Singular Kernel

As a particular example, consider the integral equation

$$f(x) + \int_0^\pi K(x,y) f(y) dy = \sin x \tag{18.3.13}$$

with the (arbitrarily chosen) nasty kernel

$$K(x,y) = \cos x \cos y \times \begin{cases} \ln(x-y) & y < x \\ \sqrt{y-x} & y \geq x \end{cases} \tag{18.3.14}$$

which has a logarithmic singularity on the left of the diagonal, combined with a square-root discontinuity on the right.

The first step is to do (analytically, in this case) the required moment integrals over the singular part of the kernel, equation (18.3.12). Since these integrals are done at a fixed value of x, we can use x as the lower limit. For any specified value of y, the required indefinite integral is then either

$$F_m(y;x) = \int_x^y s^m (s-x)^{1/2} ds = \int_0^{y-x} (x+t)^m t^{1/2} dt \qquad \text{if } y > x \tag{18.3.15}$$

or

$$F_m(y;x) = \int_x^y s^m \ln(x-s) ds = \int_0^{x-y} (x-t)^m \ln t \, dt \qquad \text{if } y < x \tag{18.3.16}$$

(where a change of variable has been made in the second equality in each case). Doing these integrals analytically (actually, we used a symbolic integration package!), we package the resulting formulas in the following routine. Note that `w`$(j+1)$ returns $F_j(y;x)$.

```
SUBROUTINE kermom(w,y,m)
    Returns in w(1:m) the first m indefinite-integral moments of one row of the singular part
    of the kernel. (For this example, m is hard-wired to be 4.) The input variable y labels the
    column, while x (in COMMON) is the row.
INTEGER m
DOUBLE PRECISION w(m),y,x,d,df,clog,x2,x3,x4
COMMON /momcom/ x
  We can take x as the lower limit of integration. Thus, we return the moment integrals either
  purely to the left or purely to the right of the diagonal.
if (y.ge.x) then
    d=y-x
    df=2.d0*sqrt(d)*d
    w(1)=df/3.d0
```

```
        w(2)=df*(x/3.d0+d/5.d0)
        w(3)=df*((x/3.d0 + 0.4d0*d)*x + d**2/7.d0)
        w(4)=df*(((x/3.d0 + 0.6d0*d)*x + 3.d0*d**2/7.d0)*x
*            + d**3/9.d0)
    else
        x2=x**2
        x3=x2*x
        x4=x2*x2
        d=x-y
        clog=log(d)
        w(1)=d*(clog-1.d0)
        w(2)=-0.25d0*(3.d0*x+y-2.d0*clog*(x+y))*d
        w(3)=(-11.d0*x3+y*(6.d0*x2+y*(3.d0*x+2.d0*y))
*            +6.d0*clog*(x3-y**3))/18.d0
        w(4)=(-25.d0*x4+y*(12.d0*x3+y*(6.d0*x2+y*
*            (4.d0*x+3.d0*y)))+12.d0*clog*(x4-y**4))/48.d0
    endif
    return
    END
```

Next, we write a routine that constructs the quadrature matrix.

```
    SUBROUTINE quadmx(a,n,np)
    INTEGER n,np,NMAX
    REAL a(np,np),PI
    DOUBLE PRECISION xx
    PARAMETER (PI=3.14159265,NMAX=257)
    COMMON /momcom/ xx
    EXTERNAL kermom
C   USES wwghts,kermom
```

Constructs in `a(1:n,1:n)` the quadrature matrix for an example Fredholm equation of the second kind. The nonsingular part of the kernel is computed within this routine, while the quadrature weights which integrate the singular part of the kernel are obtained via calls to `wwghts`. An external routine `kermom`, which supplies indefinite-integral moments of the singular part of the kernel, is passed to `wwghts`.

```
    INTEGER j,k
    REAL h,wt(NMAX),x,cx,y
    h=PI/(n-1)
    do 12 j=1,n
        x=(j-1)*h
        xx=x                          Put x in COMMON for use by kermom.
        call wwghts(wt,n,h,kermom)
        cx=cos(x)                     Part of nonsingular kernel.
        do 11 k=1,n
            y=(k-1)*h
            a(j,k)=wt(k)*cx*cos(y)    Put together all the pieces of the kernel.
        enddo 11
        a(j,j)=a(j,j)+1.              Since equation of the second kind, there is diagonal
    enddo 12                              piece independent of h.
    return
    END
```

Finally, we solve the linear system for any particular right-hand side, here $\sin x$.

```
    PROGRAM fredex
    INTEGER NMAX
    REAL PI
    PARAMETER (NMAX=100,PI=3.14159265)
    INTEGER indx(NMAX),j,n
    REAL a(NMAX,NMAX),g(NMAX),x,d
C   USES quadmx,ludcmp,lubksb
```

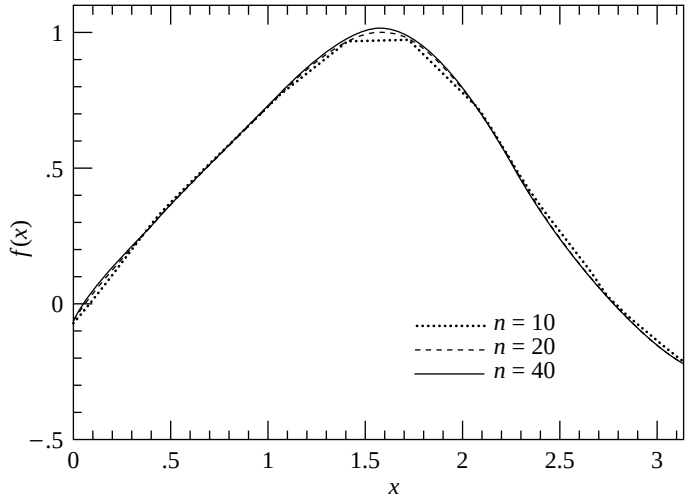


Figure 18.3.1. Solution of the example integral equation (18.3.14) with grid sizes $N = 10$, 20, and 40. The tabulated solution values have been connected by straight lines; in practice one would interpolate a small N solution more smoothly.

This sample program shows how to solve a Fredholm equation of the second kind using the product Nystrom method and a quadrature rule especially constructed for a particular, singular, kernel.

```
n=40                                  Here the size of the grid is specified.
call quadmx(a,n,NMAX)                 Make the quadrature matrix; all the action is here.
call ludcmp(a,n,NMAX,indx,d)          Decompose the matrix.
do 11 j=1,n                           Construct the right hand side, here sin x.
    x=(j-1)*PI/(n-1)
    g(j)=sin(x)
enddo 11
call lubksb(a,n,NMAX,indx,g)          Backsubstitute.
do 12 j=1,n                           Write out the solution.
    x=(j-1)*PI/(n-1)
    write (*,*) j,x,g(j)
enddo 12
write (*,*) 'normal completion'
END
```

With $N = 40$, this program gives accuracy at about the 10^{-5} level. The accuracy increases as N^4 (as it should for our Simpson-order quadrature scheme) *despite* the highly singular kernel. Figure 18.3.1 shows the solution obtained, also plotting the solution for smaller values of N, which are themselves seen to be remarkably faithful. Notice that the solution is smooth, even though the kernel is singular, a common occurrence.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [1]

Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [3]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.). [4]

18.4 Inverse Problems and the Use of A Priori Information

Later discussion will be facilitated by some preliminary mention of a couple of mathematical points. Suppose that $\mathbf{u}$ is an "unknown" vector that we plan to determine by some minimization principle. Let $\mathcal{A}[\mathbf{u}] > 0$ and $\mathcal{B}[\mathbf{u}] > 0$ be two positive functionals of $\mathbf{u}$, so that we can try to determine $\mathbf{u}$ by either

$$\text{minimize:} \quad \mathcal{A}[\mathbf{u}] \qquad \text{or} \qquad \text{minimize:} \quad \mathcal{B}[\mathbf{u}] \tag{18.4.1}$$

(Of course these will generally give different answers for $\mathbf{u}$.) As another possibility, now suppose that we want to minimize $\mathcal{A}[\mathbf{u}]$ subject to the *constraint* that $\mathcal{B}[\mathbf{u}]$ have some particular value, say b. The method of Lagrange multipliers gives the variation

$$\frac{\delta}{\delta \mathbf{u}} \{\mathcal{A}[\mathbf{u}] + \lambda_1(\mathcal{B}[\mathbf{u}] - b)\} = \frac{\delta}{\delta \mathbf{u}} (\mathcal{A}[\mathbf{u}] + \lambda_1 \mathcal{B}[\mathbf{u}]) = 0 \tag{18.4.2}$$

where λ_1 is a Lagrange multiplier. Notice that b is absent in the second equality, since it doesn't depend on $\mathbf{u}$.

Next, suppose that we change our minds and decide to minimize $\mathcal{B}[\mathbf{u}]$ subject to the constraint that $\mathcal{A}[\mathbf{u}]$ have a particular value, a. Instead of equation (18.4.2) we have

$$\frac{\delta}{\delta \mathbf{u}} \{\mathcal{B}[\mathbf{u}] + \lambda_2(\mathcal{A}[\mathbf{u}] - a)\} = \frac{\delta}{\delta \mathbf{u}} (\mathcal{B}[\mathbf{u}] + \lambda_2 \mathcal{A}[\mathbf{u}]) = 0 \tag{18.4.3}$$

with, this time, λ_2 the Lagrange multiplier. Multiplying equation (18.4.3) by the constant $1/\lambda_2$, and identifying $1/\lambda_2$ with λ_1, we see that the actual variations are exactly the same in the two cases. Both cases will yield the same one-parameter family of solutions, say, $\mathbf{u}(\lambda_1)$. As λ_1 varies from 0 to ∞, the solution $\mathbf{u}(\lambda_1)$ varies along a so-called *trade-off curve* between the problem of minimizing $\mathcal{A}$ and the problem of minimizing $\mathcal{B}$. Any solution along this curve can equally well be thought of as either (i) a minimization of $\mathcal{A}$ for some constrained value of $\mathcal{B}$, or (ii) a minimization of $\mathcal{B}$ for some constrained value of $\mathcal{A}$, or (iii) a weighted minimization of the sum $\mathcal{A} + \lambda_1 \mathcal{B}$.

The second preliminary point has to do with *degenerate* minimization principles. In the example above, now suppose that $\mathcal{A}[\mathbf{u}]$ has the particular form

$$\mathcal{A}[\mathbf{u}] = |\mathbf{A} \cdot \mathbf{u} - \mathbf{c}|^2 \tag{18.4.4}$$

for some matrix $\mathbf{A}$ and vector $\mathbf{c}$. If $\mathbf{A}$ has fewer rows than columns, or if $\mathbf{A}$ is square but degenerate (has a nontrivial nullspace, see §2.6, especially Figure 2.6.1), then minimizing $\mathcal{A}[\mathbf{u}]$ will *not* give a unique solution for $\mathbf{u}$. (To see why, review §15.4, and note that for a "design matrix" $\mathbf{A}$ with fewer rows than columns, the matrix $\mathbf{A}^T \cdot \mathbf{A}$ in the normal equations 15.4.10 is degenerate.) *However*, if we add any multiple λ times a nondegenerate quadratic form $\mathcal{B}[\mathbf{u}]$, for example $\mathbf{u} \cdot \mathbf{H} \cdot \mathbf{u}$ with $\mathbf{H}$ a positive definite matrix, then minimization of $\mathcal{A}[\mathbf{u}] + \lambda \mathcal{B}[\mathbf{u}]$ *will* lead to a unique solution for $\mathbf{u}$. (The sum of two quadratic forms is itself a quadratic form, with the second piece guaranteeing nondegeneracy.)

18.4 Inverse Problems and the Use of A Priori Information

Later discussion will be facilitated by some preliminary mention of a couple of mathematical points. Suppose that $\mathbf{u}$ is an "unknown" vector that we plan to determine by some minimization principle. Let $\mathcal{A}[\mathbf{u}] > 0$ and $\mathcal{B}[\mathbf{u}] > 0$ be two positive functionals of $\mathbf{u}$, so that we can try to determine $\mathbf{u}$ by either

$$\text{minimize:} \quad \mathcal{A}[\mathbf{u}] \qquad \text{or} \qquad \text{minimize:} \quad \mathcal{B}[\mathbf{u}] \tag{18.4.1}$$

(Of course these will generally give different answers for $\mathbf{u}$.) As another possibility, now suppose that we want to minimize $\mathcal{A}[\mathbf{u}]$ subject to the *constraint* that $\mathcal{B}[\mathbf{u}]$ have some particular value, say b. The method of Lagrange multipliers gives the variation

$$\frac{\delta}{\delta \mathbf{u}} \{\mathcal{A}[\mathbf{u}] + \lambda_1(\mathcal{B}[\mathbf{u}] - b)\} = \frac{\delta}{\delta \mathbf{u}} (\mathcal{A}[\mathbf{u}] + \lambda_1 \mathcal{B}[\mathbf{u}]) = 0 \tag{18.4.2}$$

where λ_1 is a Lagrange multiplier. Notice that b is absent in the second equality, since it doesn't depend on $\mathbf{u}$.

Next, suppose that we change our minds and decide to minimize $\mathcal{B}[\mathbf{u}]$ subject to the constraint that $\mathcal{A}[\mathbf{u}]$ have a particular value, a. Instead of equation (18.4.2) we have

$$\frac{\delta}{\delta \mathbf{u}} \{\mathcal{B}[\mathbf{u}] + \lambda_2(\mathcal{A}[\mathbf{u}] - a)\} = \frac{\delta}{\delta \mathbf{u}} (\mathcal{B}[\mathbf{u}] + \lambda_2 \mathcal{A}[\mathbf{u}]) = 0 \tag{18.4.3}$$

with, this time, λ_2 the Lagrange multiplier. Multiplying equation (18.4.3) by the constant $1/\lambda_2$, and identifying $1/\lambda_2$ with λ_1, we see that the actual variations are exactly the same in the two cases. Both cases will yield the same one-parameter family of solutions, say, $\mathbf{u}(\lambda_1)$. As λ_1 varies from 0 to ∞, the solution $\mathbf{u}(\lambda_1)$ varies along a so-called *trade-off curve* between the problem of minimizing $\mathcal{A}$ and the problem of minimizing $\mathcal{B}$. Any solution along this curve can equally well be thought of as either (i) a minimization of $\mathcal{A}$ for some constrained value of $\mathcal{B}$, or (ii) a minimization of $\mathcal{B}$ for some constrained value of $\mathcal{A}$, or (iii) a weighted minimization of the sum $\mathcal{A} + \lambda_1 \mathcal{B}$.

The second preliminary point has to do with *degenerate* minimization principles. In the example above, now suppose that $\mathcal{A}[\mathbf{u}]$ has the particular form

$$\mathcal{A}[\mathbf{u}] = |\mathbf{A} \cdot \mathbf{u} - \mathbf{c}|^2 \tag{18.4.4}$$

for some matrix $\mathbf{A}$ and vector $\mathbf{c}$. If $\mathbf{A}$ has fewer rows than columns, or if $\mathbf{A}$ is square but degenerate (has a nontrivial nullspace, see §2.6, especially Figure 2.6.1), then minimizing $\mathcal{A}[\mathbf{u}]$ will *not* give a unique solution for $\mathbf{u}$. (To see why, review §15.4, and note that for a "design matrix" $\mathbf{A}$ with fewer rows than columns, the matrix $\mathbf{A}^T \cdot \mathbf{A}$ in the normal equations 15.4.10 is degenerate.) *However*, if we add any multiple λ times a nondegenerate quadratic form $\mathcal{B}[\mathbf{u}]$, for example $\mathbf{u} \cdot \mathbf{H} \cdot \mathbf{u}$ with $\mathbf{H}$ a positive definite matrix, then minimization of $\mathcal{A}[\mathbf{u}] + \lambda \mathcal{B}[\mathbf{u}]$ *will* lead to a unique solution for $\mathbf{u}$. (The sum of two quadratic forms is itself a quadratic form, with the second piece guaranteeing nondegeneracy.)

We can combine these two points, for this conclusion: When a quadratic minimization principle is combined with a quadratic constraint, and both are positive, only *one* of the two need be nondegenerate for the overall problem to be well-posed. We are now equipped to face the subject of inverse problems.

The Inverse Problem with Zeroth-Order Regularization

Suppose that $u(x)$ is some unknown or underlying (u stands for both unknown and underlying!) physical process, which we hope to determine by a set of N measurements c_i, $i = 1, 2, \ldots, N$. The relation between $u(x)$ and the c_i's is that each c_i measures a (hopefully distinct) aspect of $u(x)$ through its own linear response kernel r_i, and with its own measurement error n_i. In other words,

$$c_i \equiv s_i + n_i = \int r_i(x)u(x)dx + n_i \tag{18.4.5}$$

(compare this to equations 13.3.1 and 13.3.2). Within the assumption of linearity, this is quite a general formulation. The c_i's might approximate values of $u(x)$ at certain locations x_i, in which case $r_i(x)$ would have the form of a more or less narrow instrumental response centered around $x = x_i$. Or, the c_i's might "live" in an entirely different function space from $u(x)$, measuring different Fourier components of $u(x)$ for example.

The *inverse problem* is, given the c_i's, the $r_i(x)$'s, and perhaps some information about the errors n_i such as their covariance matrix

$$S_{ij} \equiv \text{Covar}[n_i, n_j] \tag{18.4.6}$$

how do we find a good statistical estimator of $u(x)$, call it $\widehat{u}(x)$?

It should be obvious that this is an ill-posed problem. After all, how can we reconstruct a whole function $\widehat{u}(x)$ from only a finite number of discrete values c_i? Yet, whether formally or informally, we do this all the time in science. We routinely measure "enough points" and then "draw a curve through them." In doing so, we are making some assumptions, either about the underlying function $u(x)$, or about the nature of the response functions $r_i(x)$, or both. Our purpose now is to formalize these assumptions, and to extend our abilities to cases where the measurements and underlying function live in quite different function spaces. (How do you "draw a curve" through a scattering of Fourier coefficients?)

We can't really want every point x of the function $\widehat{u}(x)$. We do want some large number M of discrete points x_μ, $\mu = 1, 2, \ldots, M$, where M is sufficiently large, and the x_μ's are sufficiently evenly spaced, that neither $u(x)$ nor $r_i(x)$ varies much between any x_μ and $x_{\mu+1}$. (Here and following we will use Greek letters like μ to denote values in the space of the underlying process, and Roman letters like i to denote values of immediate observables.) For such a dense set of x_μ's, we can replace equation (18.4.5) by a quadrature like

$$c_i = \sum_\mu R_{i\mu}u(x_\mu) + n_i \tag{18.4.7}$$

where the $N \times M$ matrix $\mathbf{R}$ has components

$$R_{i\mu} \equiv r_i(x_\mu)(x_{\mu+1} - x_{\mu-1})/2 \tag{18.4.8}$$

(or any other simple quadrature — it rarely matters which). We will view equations (18.4.5) and (18.4.7) as being equivalent for practical purposes.

How do you solve a set of equations like equation (18.4.7) for the unknown $u(x_\mu)$'s? Here is a bad way, but one that contains the germ of some correct ideas: Form a χ^2 measure of how well a model $\widehat{u}(x)$ agrees with the measured data,

$$\begin{aligned}\chi^2 &= \sum_{i=1}^{N}\sum_{j=1}^{N}\left[c_i - \sum_{\mu=1}^{M} R_{i\mu}\widehat{u}(x_\mu)\right] S_{ij}^{-1}\left[c_j - \sum_{\mu=1}^{M} R_{j\mu}\widehat{u}(x_\mu)\right] \\ &\approx \sum_{i=1}^{N}\left[\frac{c_i - \sum_{\mu=1}^{M} R_{i\mu}\widehat{u}(x_\mu)}{\sigma_i}\right]^2\end{aligned} \tag{18.4.9}$$

(compare with equation 15.1.5). Here $\mathbf{S}^{-1}$ is the inverse of the covariance matrix, and the approximate equality holds if you can neglect the off-diagonal covariances, with $\sigma_i \equiv (\text{Covar}[i,i])^{1/2}$.

Now you can use the method of singular value decomposition (SVD) in §15.4 to find the vector $\widehat{\mathbf{u}}$ that minimizes equation (18.4.9). Don't try to use the method of normal equations; since M is greater than N they will be singular, as we already discussed. The SVD process will thus surely find a large number of zero singular values, indicative of a highly non-unique solution. Among the infinity of degenerate solutions (most of them badly behaved with arbitrarily large $\widehat{u}(x_\mu)$'s) SVD will select the one with smallest $|\widehat{\mathbf{u}}|$ in the sense of

$$\sum_{\mu}[\widehat{u}(x_\mu)]^2 \quad \text{a minimum} \tag{18.4.10}$$

(look at Figure 2.6.1). This solution is often called the *principal solution*. It is a limiting case of what is called *zeroth-order regularization*, corresponding to minimizing the sum of the two positive functionals

$$\text{minimize:} \quad \chi^2[\widehat{\mathbf{u}}] + \lambda(\widehat{\mathbf{u}}\cdot\widehat{\mathbf{u}}) \tag{18.4.11}$$

in the limit of small λ. Below, we will learn how to do such minimizations, as well as more general ones, without the *ad hoc* use of SVD.

What happens if we determine $\widehat{\mathbf{u}}$ by equation (18.4.11) with a non-infinitesimal value of λ? First, note that if $M \gg N$ (many more unknowns than equations), then $\mathbf{u}$ will often have enough freedom to be able to make χ^2 (equation 18.4.9) quite unrealistically small, if not zero. In the language of §15.1, the number of degrees of freedom $\nu = N - M$, which is approximately the expected value of χ^2 when ν is large, is being driven down to zero (and, not meaningfully, beyond). Yet, we know that for the *true* underlying function $u(x)$, which has no adjustable parameters, the number of degrees of freedom and the expected value of χ^2 should be about $\nu \approx N$.

Increasing λ pulls the solution away from minimizing χ^2 in favor of minimizing $\widehat{\mathbf{u}}\cdot\widehat{\mathbf{u}}$. From the preliminary discussion above, we can view this as minimizing $\widehat{\mathbf{u}}\cdot\widehat{\mathbf{u}}$ subject to the *constraint* that χ^2 have some constant nonzero value. A popular choice, in fact, is to find that value of λ which yields $\chi^2 = N$, that is, to get about as much extra regularization as a plausible value of χ^2 dictates. The resulting $\widehat{u}(x)$ is called *the solution of the inverse problem with zeroth-order regularization.*

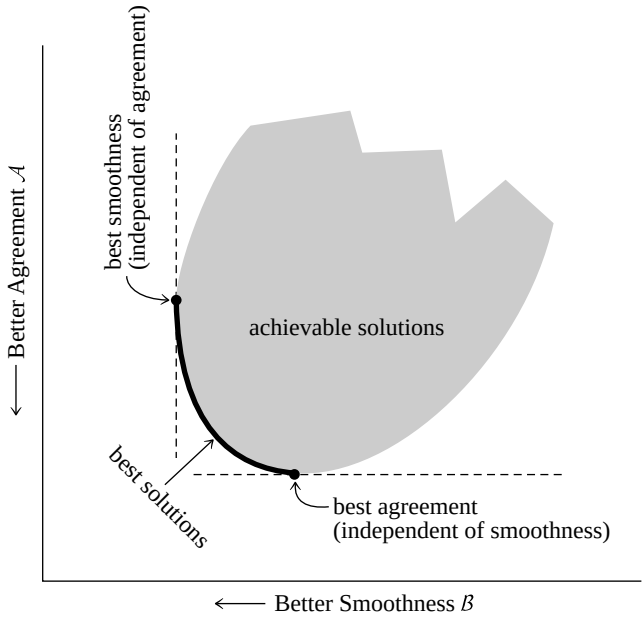


Figure 18.4.1. Almost all inverse problem methods involve a trade-off between two optimizations: agreement between data and solution, or "sharpness" of mapping between true and estimated solution (here denoted $\mathcal{A}$), and smoothness or stability of the solution (here denoted $\mathcal{B}$). Among all possible solutions, shown here schematically as the shaded region, those on the boundary connecting the unconstrained minimum of $\mathcal{A}$ and the unconstrained minimum of $\mathcal{B}$ are the "best" solutions, in the sense that every other solution is dominated by at least one solution on the curve.

The value N is actually a surrogate for any value drawn from a Gaussian distribution with mean N and standard deviation $(2N)^{1/2}$ (the asymptotic χ^2 distribution). One might equally plausibly try two values of λ, one giving $\chi^2 = N + (2N)^{1/2}$, the other $N - (2N)^{1/2}$.

Zeroth-order regularization, though dominated by better methods, demonstrates most of the basic ideas that are used in inverse problem theory. In general, there are two positive functionals, call them $\mathcal{A}$ and $\mathcal{B}$. The first, $\mathcal{A}$, measures something like the agreement of a model to the data (e.g., χ^2), or sometimes a related quantity like the "sharpness" of the mapping between the solution and the underlying function. When $\mathcal{A}$ by itself is minimized, the agreement or sharpness becomes very good (often impossibly good), but the solution becomes unstable, wildly oscillating, or in other ways unrealistic, reflecting that $\mathcal{A}$ alone typically defines a highly degenerate minimization problem.

That is where $\mathcal{B}$ comes in. It measures something like the "smoothness" of the desired solution, or sometimes a related quantity that parametrizes the stability of the solution with respect to variations in the data, or sometimes a quantity reflecting *a priori* judgments about the likelihood of a solution. $\mathcal{B}$ is called the *stabilizing functional* or *regularizing operator*. In any case, minimizing $\mathcal{B}$ by itself is supposed to give a solution that is "smooth" or "stable" or "likely" — and that has nothing at all to do with the measured data.

The single central idea in inverse theory is the prescription

$$\text{minimize:} \quad \mathcal{A} + \lambda\mathcal{B} \tag{18.4.12}$$

for various values of $0 < \lambda < \infty$ along the so-called trade-off curve (see Figure 18.4.1), and then to settle on a "best" value of λ by one or another criterion, ranging from fairly objective (e.g., making $\chi^2 = N$) to entirely subjective. Successful methods, several of which we will now describe, differ as to their choices of $\mathcal{A}$ and $\mathcal{B}$, as to whether the prescription (18.4.12) yields linear or nonlinear equations, as to their recommended method for selecting a final λ, and as to their practicality for computer-intensive two-dimensional problems like image processing.

They also differ as to the philosophical baggage that they (or rather, their proponents) carry. We have thus far avoided the word "Bayesian." (Courts have consistently held that academic license does not extend to shouting "Bayesian" in a crowded lecture hall.) But it is hard, nor have we any wish, to disguise the fact that $\mathcal{B}$ has something to do with *a priori* expectation, or knowledge, of a solution, while $\mathcal{A}$ has something to do with *a posteriori* knowledge. The constant λ adjudicates a delicate compromise between the two. Some inverse methods have acquired a more Bayesian stamp than others, but we think that this is purely an accident of history. An outsider looking only at the equations that are actually solved, and not at the accompanying philosophical justifications, would have a difficult time separating the so-called Bayesian methods from the so-called empirical ones, we think.

The next three sections discuss three different approaches to the problem of inversion, which have had considerable success in different fields. All three fit within the general framework that we have outlined, but they are quite different in detail and in implementation.

CITED REFERENCES AND FURTHER READING:

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger).

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier).

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley).

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64.

Frieden, B.R. 1975, in *Picture Processing and Digital Filtering*, T.S. Huang, ed. (New York: Springer-Verlag).

Tarantola, A. 1987, *Inverse Problem Theory* (Amsterdam: Elsevier).

Baumeister, J. 1987, *Stable Solution of Inverse Problems* (Braunschweig, Germany: Friedr. Vieweg & Sohn) [mathematically oriented].

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, pp. 381–387.

Jeffrey, W., and Rosner, R. 1986, *Astrophysical Journal*, vol. 310, pp. 463–472.

18.5 Linear Regularization Methods

What we will call *linear regularization* is also called the *Phillips-Twomey method* [1,2], the *constrained linear inversion method* [3], the *method of regularization* [4], and *Tikhonov-Miller regularization* [5-7]. (It probably has other names also,

The single central idea in inverse theory is the prescription

$$\text{minimize:} \quad \mathcal{A} + \lambda\mathcal{B} \tag{18.4.12}$$

for various values of $0 < \lambda < \infty$ along the so-called trade-off curve (see Figure 18.4.1), and then to settle on a "best" value of λ by one or another criterion, ranging from fairly objective (e.g., making $\chi^2 = N$) to entirely subjective. Successful methods, several of which we will now describe, differ as to their choices of $\mathcal{A}$ and $\mathcal{B}$, as to whether the prescription (18.4.12) yields linear or nonlinear equations, as to their recommended method for selecting a final λ, and as to their practicality for computer-intensive two-dimensional problems like image processing.

They also differ as to the philosophical baggage that they (or rather, their proponents) carry. We have thus far avoided the word "Bayesian." (Courts have consistently held that academic license does not extend to shouting "Bayesian" in a crowded lecture hall.) But it is hard, nor have we any wish, to disguise the fact that $\mathcal{B}$ has something to do with *a priori* expectation, or knowledge, of a solution, while $\mathcal{A}$ has something to do with *a posteriori* knowledge. The constant λ adjudicates a delicate compromise between the two. Some inverse methods have acquired a more Bayesian stamp than others, but we think that this is purely an accident of history. An outsider looking only at the equations that are actually solved, and not at the accompanying philosophical justifications, would have a difficult time separating the so-called Bayesian methods from the so-called empirical ones, we think.

The next three sections discuss three different approaches to the problem of inversion, which have had considerable success in different fields. All three fit within the general framework that we have outlined, but they are quite different in detail and in implementation.

CITED REFERENCES AND FURTHER READING:

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger).

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier).

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley).

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64.

Frieden, B.R. 1975, in *Picture Processing and Digital Filtering*, T.S. Huang, ed. (New York: Springer-Verlag).

Tarantola, A. 1987, *Inverse Problem Theory* (Amsterdam: Elsevier).

Baumeister, J. 1987, *Stable Solution of Inverse Problems* (Braunschweig, Germany: Friedr. Vieweg & Sohn) [mathematically oriented].

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, pp. 381–387.

Jeffrey, W., and Rosner, R. 1986, *Astrophysical Journal*, vol. 310, pp. 463–472.

18.5 Linear Regularization Methods

What we will call *linear regularization* is also called the *Phillips-Twomey method* [1,2], the *constrained linear inversion method* [3], the *method of regularization* [4], and *Tikhonov-Miller regularization* [5-7]. (It probably has other names also,

since it is so obviously a good idea.) In its simplest form, the method is an immediate generalization of zeroth-order regularization (equation 18.4.11, above). As before, the functional $\mathcal{A}$ is taken to be the χ^2 deviation, equation (18.4.9), but the functional $\mathcal{B}$ is replaced by more sophisticated measures of smoothness that derive from first or higher derivatives.

For example, suppose that your *a priori* belief is that a credible $u(x)$ is not too different from a constant. Then a reasonable functional to minimize is

$$\mathcal{B} \propto \int [\widehat{u}'(x)]^2 dx \propto \sum_{\mu=1}^{M-1} [\widehat{u}_\mu - \widehat{u}_{\mu+1}]^2 \tag{18.5.1}$$

since it is nonnegative and equal to zero only when $\widehat{u}(x)$ is constant. Here $\widehat{u}_\mu \equiv \widehat{u}(x_\mu)$, and the second equality (proportionality) assumes that the x_μ's are uniformly spaced. We can write the second form of $\mathcal{B}$ as

$$\mathcal{B} = |\mathbf{B} \cdot \widehat{\mathbf{u}}|^2 = \widehat{\mathbf{u}} \cdot (\mathbf{B}^T \cdot \mathbf{B}) \cdot \widehat{\mathbf{u}} \equiv \widehat{\mathbf{u}} \cdot \mathbf{H} \cdot \widehat{\mathbf{u}} \tag{18.5.2}$$

where $\widehat{\mathbf{u}}$ is the vector of components $\widehat{u}_\mu$, $\mu = 1, \ldots, M$, $\mathbf{B}$ is the $(M-1) \times M$ first difference matrix

$$\mathbf{B} = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.3}$$

and $\mathbf{H}$ is the $M \times M$ matrix

$$\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.4}$$

Note that $\mathbf{B}$ has one fewer row than column. It follows that the symmetric $\mathbf{H}$ is degenerate; it has exactly one zero eigenvalue corresponding to the *value* of a constant function, any one of which makes $\mathcal{B}$ exactly zero.

If, just as in §15.4, we write

$$A_{i\mu} \equiv R_{i\mu}/\sigma_i \qquad b_i \equiv c_i/\sigma_i \tag{18.5.5}$$

then, using equation (18.4.9), the minimization principle (18.4.12) is

$$\text{minimize:} \quad \mathcal{A} + \lambda\mathcal{B} = |\mathbf{A} \cdot \widehat{\mathbf{u}} - \mathbf{b}|^2 + \lambda \widehat{\mathbf{u}} \cdot \mathbf{H} \cdot \widehat{\mathbf{u}} \tag{18.5.6}$$

This can readily be reduced to a linear set of *normal equations*, just as in §15.4: The components $\widehat{u}_\mu$ of the solution satisfy the set of M equations in M unknowns,

$$\sum_\rho \left[\left(\sum_i A_{i\mu} A_{i\rho} \right) + \lambda H_{\mu\rho} \right] \widehat{u}_\rho = \sum_i A_{i\mu} b_i \qquad \mu = 1, 2, \ldots, M \tag{18.5.7}$$

or, in vector notation,

$$(\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H}) \cdot \widehat{\mathbf{u}} = \mathbf{A}^T \cdot \mathbf{b} \tag{18.5.8}$$

Equations (18.5.7) or (18.5.8) can be solved by the standard techniques of Chapter 2, e.g., LU decomposition. The usual warnings about normal equations being ill-conditioned do not apply, since the whole purpose of the λ term is to cure that same ill-conditioning. Note, however, that the λ term *by itself* is ill-conditioned, since it does not select a preferred constant value. You hope your data can at least do *that*!

Although inversion of the matrix $(\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H})$ is not generally the best way to solve for $\widehat{\mathbf{u}}$, let us digress to write the solution to equation (18.5.8) schematically as

$$\widehat{\mathbf{u}} = \left(\frac{1}{\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H}} \cdot \mathbf{A}^T \cdot \mathbf{A} \right) \mathbf{A}^{-1} \cdot \mathbf{b} \qquad \text{(schematic only!)} \tag{18.5.9}$$

where the identity matrix in the form $\mathbf{A} \cdot \mathbf{A}^{-1}$ has been inserted. This is schematic not only because the matrix inverse is fancifully written as a denominator, but also because, in general, the inverse matrix $\mathbf{A}^{-1}$ does not exist. However, it is illuminating to compare equation (18.5.9) with equation (13.3.6) for optimal or Wiener filtering, or with equation (13.6.6) for general linear prediction. One sees that $\mathbf{A}^T \cdot \mathbf{A}$ plays the role of S^2, the signal power or autocorrelation, while $\lambda \mathbf{H}$ plays the role of N^2, the noise power or autocorrelation. The term in parentheses in equation (18.5.9) is something like an optimal filter, whose effect is to pass the ill-posed inverse $\mathbf{A}^{-1} \cdot \mathbf{b}$ through unmodified when $\mathbf{A}^T \cdot \mathbf{A}$ is sufficiently large, but to suppress it when $\mathbf{A}^T \cdot \mathbf{A}$ is small.

The above choices of $\mathbf{B}$ and $\mathbf{H}$ are only the simplest in an obvious sequence of derivatives. If your *a priori* belief is that a *linear* function is a good approximation to $u(x)$, then minimize

$$\mathcal{B} \propto \int [\widehat{u}''(x)]^2 dx \propto \sum_{\mu=1}^{M-2} [-\widehat{u}_\mu + 2\widehat{u}_{\mu+1} - \widehat{u}_{\mu+2}]^2 \tag{18.5.10}$$

implying

$$\mathbf{B} = \begin{pmatrix} -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \end{pmatrix} \tag{18.5.11}$$

and

$$\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B} = \begin{pmatrix} 1 & -2 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -2 & 5 & -4 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 1 & -4 & 6 & -4 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 1 & -4 & 5 & -2 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1 & -2 & 1 \end{pmatrix} \tag{18.5.12}$$

This **H** has two zero eigenvalues, corresponding to the two undetermined parameters of a linear function.

If your *a priori* belief is that a *quadratic* function is preferable, then minimize

$$\mathcal{B} \propto \int [\widehat{u}'''(x)]^2 dx \propto \sum_{\mu=1}^{M-3} [-\widehat{u}_\mu + 3\widehat{u}_{\mu+1} - 3\widehat{u}_{\mu+2} + \widehat{u}_{\mu+3}]^2 \tag{18.5.13}$$

with

$$\mathbf{B} = \begin{pmatrix} -1 & 3 & -3 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 3 & -3 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & -1 & 3 & -3 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 3 & -3 & 1 \end{pmatrix} \tag{18.5.14}$$

and now

$$\mathbf{H} = \begin{pmatrix} 1 & -3 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -3 & 10 & -12 & 6 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 3 & -12 & 19 & -15 & 6 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & \cdots & 0 \\ \vdots & & & & & \ddots & & & & & \vdots \\ 0 & \cdots & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 \\ 0 & \cdots & 0 & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 6 & -15 & 19 & -12 & 3 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 6 & -12 & 10 & -3 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -3 & 1 \end{pmatrix} \tag{18.5.15}$$

(We'll leave the calculation of cubics and above to the compulsive reader.)

Notice that you can regularize with "closeness to a differential equation," if you want. Just pick **B** to be the appropriate sum of finite-difference operators (the coefficients can depend on x), and calculate $\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B}$. You don't need to know the values of your boundary conditions, since **B** can have fewer rows than columns, as above; hopefully, your data will determine them. Of course, if you do know some boundary conditions, you can build these into **B** too.

With all the proportionality signs above, you may have lost track of what actual value of λ to try first. A simple trick for at least getting "on the map" is to first try

$$\lambda = \text{Tr}(\mathbf{A}^T \cdot \mathbf{A})/\text{Tr}(\mathbf{H}) \tag{18.5.16}$$

where Tr is the trace of the matrix (sum of diagonal components). This choice will tend to make the two parts of the minimization have comparable weights, and you can adjust from there.

As for what is the "correct" value of λ, an objective criterion, if you know your errors σ_i with reasonable accuracy, is to make χ^2 (that is, $|\mathbf{A} \cdot \widehat{\mathbf{u}} - \mathbf{b}|^2$) equal to N, the number of measurements. We remarked above on the twin acceptable choices $N \pm (2N)^{1/2}$. A subjective criterion is to pick any value that you like in the

range $0 < \lambda < \infty$, depending on your relative degree of belief in the *a priori* and *a posteriori* evidence. (Yes, people actually do that. Don't blame us.)

Two-Dimensional Problems and Iterative Methods

Up to now our notation has been indicative of a one-dimensional problem, finding $\widehat{u}(x)$ or $\widehat{u}_\mu = \widehat{u}(x_\mu)$. However, all of the discussion easily generalizes to the problem of estimating a two-dimensional set of unknowns $\widehat{u}_{\mu\kappa}$, $\mu = 1, \ldots, M$, $\kappa = 1, \ldots, K$, corresponding, say, to the pixel intensities of a measured image. In this case, equation (18.5.8) is still the one we want to solve.

In image processing, it is usual to have the same number of input pixels in a measured "raw" or "dirty" image as desired "clean" pixels in the processed output image, so the matrices **R** and **A** (equation 18.5.5) are square and of size $MK \times MK$. **A** is typically much too large to represent as a full matrix, but often it is either (i) sparse, with coefficients blurring an underlying pixel (i, j) only into measurements $(i \pm \text{few}, j \pm \text{few})$, or (ii) translationally invariant, so that $A_{(i,j)(\mu,\nu)} = A(i-\mu, j-\nu)$. Both of these situations lead to tractable problems.

In the case of translational invariance, fast Fourier transforms (FFTs) are the obvious method of choice. The general linear relation between underlying function and measured values (18.4.7) now becomes a discrete convolution like equation (13.1.1). If **k** denotes a two-dimensional wave-vector, then the two-dimensional FFT takes us back and forth between the transform pairs

$$A(i-\mu, j-\nu) \iff \widetilde{\mathbf{A}}(\mathbf{k}) \qquad b_{(i,j)} \iff \widetilde{b}(\mathbf{k}) \qquad \widehat{u}_{(i,j)} \iff \widetilde{u}(\mathbf{k}) \tag{18.5.17}$$

We also need a regularization or smoothing operator **B** and the derived $\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B}$. One popular choice for **B** is the five-point finite-difference approximation of the Laplacian operator, that is, the difference between the value of each point and the average of its four Cartesian neighbors. In Fourier space, this choice implies,

$$\begin{aligned} \widetilde{B}(\mathbf{k}) &\propto \sin^2(\pi k_1/M)\sin^2(\pi k_2/K) \\ \widetilde{H}(\mathbf{k}) &\propto \sin^4(\pi k_1/M)\sin^4(\pi k_2/K) \end{aligned} \tag{18.5.18}$$

In Fourier space, equation (18.5.7) is merely algebraic, with solution

$$\widetilde{u}(\mathbf{k}) = \frac{\widetilde{A}^*(\mathbf{k})\widetilde{b}(\mathbf{k})}{|\widetilde{A}(\mathbf{k})|^2 + \lambda \widetilde{H}(\mathbf{k})} \tag{18.5.19}$$

where asterisk denotes complex conjugation. You can make use of the FFT routines for real data in §12.5.

Turn now to the case where **A** is not translationally invariant. Direct solution of (18.5.8) is now hopeless, since the matrix **A** is just too large. We need some kind of iterative scheme.

One way to proceed is to use the full machinery of the conjugate gradient method in §10.6 to find the minimum of $\mathcal{A} + \lambda\mathcal{B}$, equation (18.5.6). Of the various methods in Chapter 10, conjugate gradient is the unique best choice because (i) it does not require storage of a Hessian matrix, which would be infeasible here,

and (ii) it does exploit gradient information, which we can readily compute: The gradient of equation (18.5.6) is

$$\nabla(\mathcal{A}+\lambda\mathcal{B}) = 2[(\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H})\cdot\widehat{\mathbf{u}}-\mathbf{A}^T\cdot\mathbf{b}] \tag{18.5.20}$$

(cf. 18.5.8). Evaluation of both the function and the gradient should of course take advantage of the sparsity of $\mathbf{A}$, for example via the routines `sprsax` and `sprstx` in §2.7. We will discuss the conjugate gradient technique further in §18.7, in the context of the (nonlinear) maximum entropy method. Some of that discussion can apply here as well.

The conjugate gradient method notwithstanding, application of the unsophisticated steepest descent method (see §10.6) can sometimes produce useful results, particularly when combined with projections onto convex sets (see below). If the solution after k iterations is denoted $\widehat{\mathbf{u}}^{(k)}$, then after $k+1$ iterations we have

$$\widehat{\mathbf{u}}^{(k+1)} = [\mathbf{1}-\epsilon(\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H})]\cdot\widehat{\mathbf{u}}^{(k)}+\epsilon\mathbf{A}^T\cdot\mathbf{b} \tag{18.5.21}$$

Here ϵ is a parameter that dictates how far to move in the downhill gradient direction. The method converges when ϵ is small enough, in particular satisfying

$$0<\epsilon<\frac{2}{\text{max eigenvalue }(\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H})} \tag{18.5.22}$$

There exist complicated schemes for finding optimal values or sequences for ϵ, see [7]; or, one can adopt an experimental approach, evaluating (18.5.6) to be sure that downhill steps are in fact being taken.

In those image processing problems where the final measure of success is somewhat subjective (e.g., "how good does the picture look?"), iteration (18.5.21) sometimes produces significantly improved images long before convergence is achieved. This probably accounts for much of its use, since its mathematical convergence is extremely slow. In fact, (18.5.21) can be used with $\mathbf{H}=0$, in which case the solution is not regularized at all, and full convergence would be disastrous! This is called *Van Cittert's method* and goes back to the 1930s. A number of iterations the order of 1000 is not uncommon [7].

Deterministic Constraints: Projections onto Convex Sets

A set of possible underlying functions (or images) $\{\widehat{\mathbf{u}}\}$ is said to be *convex* if, for any two elements $\widehat{\mathbf{u}}_a$ and $\widehat{\mathbf{u}}_b$ in the set, all the linearly interpolated combinations

$$(1-\eta)\widehat{\mathbf{u}}_a+\eta\widehat{\mathbf{u}}_b \qquad 0\le\eta\le 1 \tag{18.5.23}$$

are also in the set. Many *deterministic constraints* that one might want to impose on the solution $\widehat{\mathbf{u}}$ to an inverse problem in fact define convex sets, for example:

- positivity
- compact support (i.e., zero value outside of a certain region)

- known bounds (i.e., $u_L(x) \le \hat{u}(x) \le u_U(x)$ for specified functions u_L and u_U).

(In this last case, the bounds might be related to an initial estimate and its error bars, e.g., $\hat{u}_0(x) \pm \gamma\sigma(x)$, where γ is of order 1 or 2.) Notice that these, and similar, constraints can be either in the image space, or in the Fourier transform space, or (in fact) in the space of any linear transformation of $\hat{\mathbf{u}}$.

If C_i is a convex set, then $\mathcal{P}_i$ is called a *nonexpansive projection operator* onto that set if (i) $\mathcal{P}_i$ leaves unchanged any $\hat{\mathbf{u}}$ already in C_i, and (ii) $\mathcal{P}_i$ maps any $\hat{\mathbf{u}}$ outside C_i to the *closest* element of C_i, in the sense that

$$|\mathcal{P}_i\hat{\mathbf{u}} - \hat{\mathbf{u}}| \le |\hat{\mathbf{u}}_a - \hat{\mathbf{u}}| \quad \text{for all } \hat{\mathbf{u}}_a \text{ in } C_i \tag{18.5.24}$$

While this definition sounds complicated, examples are very simple: A nonexpansive projection onto the set of positive $\hat{\mathbf{u}}$'s is "set all negative components of $\hat{\mathbf{u}}$ equal to zero." A nonexpansive projection onto the set of $\hat{u}(x)$'s bounded by $u_L(x) \le \hat{u}(x) \le u_U(x)$ is "set all values less than the lower bound equal to that bound, and set all values greater than the upper bound equal to *that* bound." A nonexpansive projection onto functions with compact support is "zero the values outside of the region of support."

The usefulness of these definitions is the following remarkable theorem: Let C be the intersection of m convex sets $C_1, C_2, \ldots, C_m$. Then the iteration

$$\hat{\mathbf{u}}^{(k+1)} = (\mathcal{P}_1\mathcal{P}_2\cdots\mathcal{P}_m)\hat{\mathbf{u}}^{(k)} \tag{18.5.25}$$

will converge to C from all starting points, as $k \to \infty$. Also, if C is empty (there is no intersection), then the iteration will have no limit point. Application of this theorem is called the *method of projections onto convex sets* or sometimes *POCS* [7].

A generalization of the POCS theorem is that the $\mathcal{P}_i$'s can be replaced by a set of $\mathcal{T}_i$'s,

$$\mathcal{T}_i \equiv \mathbf{1} + \beta_i(\mathcal{P}_i - \mathbf{1}) \qquad 0 < \beta_i < 2 \tag{18.5.26}$$

A well-chosen set of β_i's can accelerate the convergence to the intersection set C.

Some inverse problems can be completely solved by iteration (18.5.25) alone! For example, a problem that occurs in both astronomical imaging and X-ray diffraction work is to recover an image given only the *modulus* of its Fourier transform (equivalent to its power spectrum or autocorrelation) and not the *phase*. Here three convex sets can be utilized: the set of all images whose Fourier transform has the specified modulus to within specified error bounds; the set of all positive images; and the set of all images with zero intensity outside of some specified region. In this case the POCS iteration (18.5.25) cycles among these three, imposing each constraint in turn; FFTs are used to get in and out of Fourier space each time the Fourier constraint is imposed.

The specific application of POCS to constraints alternately in the spatial and Fourier domains is also known as the *Gerchberg-Saxton* algorithm [8]. While this algorithm is non-expansive, and is frequently convergent in practice, it has not been proved to converge in all cases [9]. In the phase-retrieval problem mentioned above, the algorithm often "gets stuck" on a plateau for many iterations before making sudden, dramatic improvements. As many as 10^4 to 10^5 iterations are sometimes

necessary. (For "unsticking" procedures, see [10].) The uniqueness of the solution is also not well understood, although for two-dimensional images of reasonable complexity it is believed to be unique.

Deterministic constraints can be incorporated, via projection operators, into iterative methods of linear regularization. In particular, rearranging terms somewhat, we can write the iteration (18.5.21) as

$$\widehat{\mathbf{u}}^{(k+1)} = [\mathbf{1} - \epsilon\lambda\mathbf{H}] \cdot \widehat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A} \cdot \widehat{\mathbf{u}}^{(k)}) \tag{18.5.27}$$

If the iteration is modified by the insertion of projection operators at each step

$$\widehat{\mathbf{u}}^{(k+1)} = (\mathcal{P}_1\mathcal{P}_2 \cdots \mathcal{P}_m)[\mathbf{1} - \epsilon\lambda\mathbf{H}] \cdot \widehat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A} \cdot \widehat{\mathbf{u}}^{(k)}) \tag{18.5.28}$$

(or, instead of $\mathcal{P}_i$'s, the $\mathcal{T}_i$ operators of equation 18.5.26), then it can be shown that the convergence condition (18.5.22) is unmodified, and the iteration will converge to minimize the quadratic functional (18.5.6) subject to the desired nonlinear deterministic constraints. See [7] for references to more sophisticated, and faster converging, iterations along these lines.

CITED REFERENCES AND FURTHER READING:

Phillips, D.L. 1962, *Journal of the Association for Computing Machinery*, vol. 9, pp. 84–97. [1]

Twomey, S. 1963, *Journal of the Association for Computing Machinery*, vol. 10, pp. 97–101. [2]

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier). [3]

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger). [4]

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley). [5]

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Miller, K. 1970, *SIAM Journal on Mathematical Analysis*, vol. 1, pp. 52–74. [6]

Schafer, R.W., Mersereau, R.M., and Richards, M.A. 1981, *Proceedings of the IEEE*, vol. 69, pp. 432–450.

Biemond, J., Lagendijk, R.L., and Mersereau, R.M. 1990, *Proceedings of the IEEE*, vol. 78, pp. 856–883. [7]

Gerchberg, R.W., and Saxton, W.O. 1972, *Optik*, vol. 35, pp. 237–246. [8]

Fienup, J.R. 1982, *Applied Optics*, vol. 15, pp. 2758–2769. [9]

Fienup, J.R., and Wackerman, C.C. 1986, *Journal of the Optical Society of America A*, vol. 3, pp. 1897–1907. [10]

18.6 Backus-Gilbert Method

The *Backus-Gilbert method* [1,2] (see, e.g., [3] or [4] for summaries) differs from other regularization methods in the nature of its functionals $\mathcal{A}$ and $\mathcal{B}$. For $\mathcal{B}$, the method seeks to maximize the *stability* of the solution $\widehat{u}(x)$ rather than, in the first instance, its smoothness. That is,

$$\mathcal{B} \equiv \mathrm{Var}[\widehat{u}(x)] \tag{18.6.1}$$

necessary. (For "unsticking" procedures, see [10].) The uniqueness of the solution is also not well understood, although for two-dimensional images of reasonable complexity it is believed to be unique.

Deterministic constraints can be incorporated, via projection operators, into iterative methods of linear regularization. In particular, rearranging terms somewhat, we can write the iteration (18.5.21) as

$$\widehat{\mathbf{u}}^{(k+1)} = [\mathbf{1} - \epsilon\lambda\mathbf{H}] \cdot \widehat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A} \cdot \widehat{\mathbf{u}}^{(k)}) \tag{18.5.27}$$

If the iteration is modified by the insertion of projection operators at each step

$$\widehat{\mathbf{u}}^{(k+1)} = (\mathcal{P}_1\mathcal{P}_2 \cdots \mathcal{P}_m)[\mathbf{1} - \epsilon\lambda\mathbf{H}] \cdot \widehat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A} \cdot \widehat{\mathbf{u}}^{(k)}) \tag{18.5.28}$$

(or, instead of $\mathcal{P}_i$'s, the $\mathcal{T}_i$ operators of equation 18.5.26), then it can be shown that the convergence condition (18.5.22) is unmodified, and the iteration will converge to minimize the quadratic functional (18.5.6) subject to the desired nonlinear deterministic constraints. See [7] for references to more sophisticated, and faster converging, iterations along these lines.

CITED REFERENCES AND FURTHER READING:

Phillips, D.L. 1962, *Journal of the Association for Computing Machinery*, vol. 9, pp. 84–97. [1]

Twomey, S. 1963, *Journal of the Association for Computing Machinery*, vol. 10, pp. 97–101. [2]

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier). [3]

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger). [4]

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley). [5]

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Miller, K. 1970, *SIAM Journal on Mathematical Analysis*, vol. 1, pp. 52–74. [6]

Schafer, R.W., Mersereau, R.M., and Richards, M.A. 1981, *Proceedings of the IEEE*, vol. 69, pp. 432–450.

Biemond, J., Lagendijk, R.L., and Mersereau, R.M. 1990, *Proceedings of the IEEE*, vol. 78, pp. 856–883. [7]

Gerchberg, R.W., and Saxton, W.O. 1972, *Optik*, vol. 35, pp. 237–246. [8]

Fienup, J.R. 1982, *Applied Optics*, vol. 15, pp. 2758–2769. [9]

Fienup, J.R., and Wackerman, C.C. 1986, *Journal of the Optical Society of America A*, vol. 3, pp. 1897–1907. [10]

18.6 Backus-Gilbert Method

The *Backus-Gilbert method* [1,2] (see, e.g., [3] or [4] for summaries) differs from other regularization methods in the nature of its functionals $\mathcal{A}$ and $\mathcal{B}$. For $\mathcal{B}$, the method seeks to maximize the *stability* of the solution $\widehat{u}(x)$ rather than, in the first instance, its smoothness. That is,

$$\mathcal{B} \equiv \mathrm{Var}[\widehat{u}(x)] \tag{18.6.1}$$

is used as a measure of how much the solution $\widehat{u}(x)$ varies as the data vary within their measurement errors. Note that this variance is not the expected deviation of $\widehat{u}(x)$ from the true $u(x)$ — that will be constrained by $\mathcal{A}$ — but rather measures the expected experiment-to-experiment scatter among estimates $\widehat{u}(x)$ if the whole experiment were to be repeated many times.

For $\mathcal{A}$ the Backus-Gilbert method looks at the relationship between the solution $\widehat{u}(x)$ and the true function $u(x)$, and seeks to make the mapping between these as close to the identity map as possible in the limit of error-free data. The method is linear, so the relationship between $\widehat{u}(x)$ and $u(x)$ can be written as

$$\widehat{u}(x) = \int \widehat{\delta}(x, x')u(x')dx' \tag{18.6.2}$$

for some so-called *resolution function* or *averaging kernel* $\widehat{\delta}(x, x')$. The Backus-Gilbert method seeks to minimize the width or *spread* of $\widehat{\delta}$ (that is, maximize the resolving power). $\mathcal{A}$ is chosen to be some positive measure of the spread.

While Backus-Gilbert's philosophy is thus rather different from that of Phillips-Twomey and related methods, in practice the differences between the methods are less than one might think. A *stable* solution is almost inevitably bound to be *smooth*: The wild, unstable oscillations that result from an unregularized solution are always exquisitely sensitive to small changes in the data. Likewise, making $\widehat{u}(x)$ close to $u(x)$ inevitably will bring error-free data into agreement with the model. Thus $\mathcal{A}$ and $\mathcal{B}$ play roles closely analogous to their corresponding roles in the previous two sections.

The principal advantage of the Backus-Gilbert formulation is that it gives good control over just those properties that it seeks to measure, namely stability and resolving power. Moreover, in the Backus-Gilbert method, the choice of λ (playing its usual role of compromise between $\mathcal{A}$ and $\mathcal{B}$) is conventionally made, or at least can easily be made, *before* any actual data are processed. One's uneasiness at making a *post hoc*, and therefore potentially subjectively biased, choice of λ is thus removed. Backus-Gilbert is often recommended as the method of choice for designing, and predicting the performance of, experiments that require data inversion.

Let's see how this all works. Starting with equation (18.4.5),

$$c_i \equiv s_i + n_i = \int r_i(x)u(x)dx + n_i \tag{18.6.3}$$

and building in linearity from the start, we seek a set of *inverse response kernels* $q_i(x)$ such that

$$\widehat{u}(x) = \sum_i q_i(x)c_i \tag{18.6.4}$$

is the desired estimator of $u(x)$. It is useful to define the integrals of the response kernels for each data point,

$$R_i \equiv \int r_i(x)dx \tag{18.6.5}$$

Substituting equation (18.6.4) into equation (18.6.3), and comparing with equation (18.6.2), we see that

$$\widehat{\delta}(x,x') = \sum_i q_i(x) r_i(x') \tag{18.6.6}$$

We can require this averaging kernel to have unit area at every x, giving

$$1 = \int \widehat{\delta}(x,x')dx' = \sum_i q_i(x) \int r_i(x')dx' = \sum_i q_i(x) R_i \equiv \mathbf{q}(x)\cdot\mathbf{R} \tag{18.6.7}$$

where $\mathbf{q}(x)$ and $\mathbf{R}$ are each vectors of length N, the number of measurements.

Standard propagation of errors, and equation (18.6.1), give

$$\mathcal{B} = \mathrm{Var}[\widehat{u}(x)] = \sum_i \sum_j q_i(x) S_{ij} q_j(x) = \mathbf{q}(x)\cdot\mathbf{S}\cdot\mathbf{q}(x) \tag{18.6.8}$$

where S_{ij} is the covariance matrix (equation 18.4.6). If one can neglect off-diagonal covariances (as when the errors on the c_i's are independent), then $S_{ij} = \delta_{ij}\sigma_i^2$ is diagonal.

We now need to define a measure of the width or spread of $\widehat{\delta}(x,x')$ at each value of x. While many choices are possible, Backus and Gilbert choose the second moment of its square. This measure becomes the functional $\mathcal{A}$,

$$\begin{aligned}\mathcal{A} \equiv w(x) &= \int (x'-x)^2 [\widehat{\delta}(x,x')]^2 dx' \\ &= \sum_i \sum_j q_i(x) W_{ij}(x) q_j(x) \equiv \mathbf{q}(x)\cdot\mathbf{W}(x)\cdot\mathbf{q}(x)\end{aligned} \tag{18.6.9}$$

where we have here used equation (18.6.6) and defined the *spread matrix* $\mathbf{W}(x)$ by

$$W_{ij}(x) \equiv \int (x'-x)^2 r_i(x') r_j(x') dx' \tag{18.6.10}$$

The functions $q_i(x)$ are now determined by the minimization principle

$$\text{minimize:}\quad \mathcal{A} + \lambda\mathcal{B} = \mathbf{q}(x)\cdot\big[\mathbf{W}(x) + \lambda\mathbf{S}\big]\cdot\mathbf{q}(x) \tag{18.6.11}$$

subject to the constraint (18.6.7) that $\mathbf{q}(x)\cdot\mathbf{R} = 1$.

The solution of equation (18.6.11) is

$$\mathbf{q}(x) = \frac{[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}}{\mathbf{R}\cdot[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}} \tag{18.6.12}$$

(Reference [4] gives an accessible proof.) For any particular data set $\mathbf{c}$ (set of measurements c_i), the solution $\widehat{u}(x)$ is thus

$$\widehat{u}(x) = \frac{\mathbf{c}\cdot[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}}{\mathbf{R}\cdot[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}} \tag{18.6.13}$$

(Don't let this notation mislead you into inverting the full matrix $\mathbf{W}(x) + \lambda\mathbf{S}$. You only need to solve for some $\mathbf{y}$ the linear system $(\mathbf{W}(x) + \lambda\mathbf{S}) \cdot \mathbf{y} = \mathbf{R}$, and then substitute $\mathbf{y}$ into both the numerators and denominators of 18.6.12 or 18.6.13.)

Equations (18.6.12) and (18.6.13) have a completely different character from the linearly regularized solutions to (18.5.7) and (18.5.8). The vectors and matrices in (18.6.12) all have size N, the number of measurements. There is no discretization of the underlying variable x, so M does not come into play at all. One solves a different $N \times N$ set of linear equations for each desired value of x. By contrast, in (18.5.8), one solves an $M \times M$ linear set, but only once. In general, the computational burden of repeatedly solving linear systems makes the Backus-Gilbert method unsuitable for other than one-dimensional problems.

How does one choose λ within the Backus-Gilbert scheme? As already mentioned, you can (in some cases *should*) make the choice *before* you see any actual data. For a given trial value of λ, and for a sequence of x's, use equation (18.6.12) to calculate $\mathbf{q}(x)$; then use equation (18.6.6) to plot the resolution functions $\widehat{\delta}(x, x')$ as a function of x'. These plots will exhibit the amplitude with which different underlying values x' contribute to the point $\widehat{u}(x)$ of your estimate. For the same value of λ, also plot the function $\sqrt{\mathrm{Var}[\widehat{u}(x)]}$ using equation (18.6.8). (You need an estimate of your measurement covariance matrix for this.)

As you change λ you will see very explicitly the trade-off between resolution and stability. Pick the value that meets your needs. You can even choose λ to be a function of x, $\lambda = \lambda(x)$, in equations (18.6.12) and (18.6.13), should you desire to do so. (This is one benefit of solving a separate set of equations for each x.) For the chosen value or values of λ, you now have a quantitative understanding of your inverse solution procedure. This can prove invaluable if — once you are processing real data — you need to judge whether a particular feature, a spike or jump for example, is genuine, and/or is actually resolved. The Backus-Gilbert method has found particular success among geophysicists, who use it to obtain information about the structure of the Earth (e.g., density run with depth) from seismic travel time data.

CITED REFERENCES AND FURTHER READING:

Backus, G.E., and Gilbert, F. 1968, *Geophysical Journal of the Royal Astronomical Society*, vol. 16, pp. 169–205. [1]

Backus, G.E., and Gilbert, F. 1970, *Philosophical Transactions of the Royal Society of London A*, vol. 266, pp. 123–192. [2]

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64. [3]

Loredo, T.J., and Epstein, R.I. 1989, *Astrophysical Journal*, vol. 336, pp. 896–919. [4]

18.7 Maximum Entropy Image Restoration

Above, we commented that the association of certain inversion methods with Bayesian arguments is more historical accident than intellectual imperative. *Maximum entropy methods*, so-called, are notorious in this regard; to summarize these methods without some, at least introductory, Bayesian invocations would be to serve a steak without the sizzle, or a sundae without the cherry. We should

(Don't let this notation mislead you into inverting the full matrix $\mathbf{W}(x) + \lambda\mathbf{S}$. You only need to solve for some $\mathbf{y}$ the linear system $(\mathbf{W}(x) + \lambda\mathbf{S}) \cdot \mathbf{y} = \mathbf{R}$, and then substitute $\mathbf{y}$ into both the numerators and denominators of 18.6.12 or 18.6.13.)

Equations (18.6.12) and (18.6.13) have a completely different character from the linearly regularized solutions to (18.5.7) and (18.5.8). The vectors and matrices in (18.6.12) all have size N, the number of measurements. There is no discretization of the underlying variable x, so M does not come into play at all. One solves a different $N \times N$ set of linear equations for each desired value of x. By contrast, in (18.5.8), one solves an $M \times M$ linear set, but only once. In general, the computational burden of repeatedly solving linear systems makes the Backus-Gilbert method unsuitable for other than one-dimensional problems.

How does one choose λ within the Backus-Gilbert scheme? As already mentioned, you can (in some cases *should*) make the choice *before* you see any actual data. For a given trial value of λ, and for a sequence of x's, use equation (18.6.12) to calculate $\mathbf{q}(x)$; then use equation (18.6.6) to plot the resolution functions $\widehat{\delta}(x, x')$ as a function of x'. These plots will exhibit the amplitude with which different underlying values x' contribute to the point $\widehat{u}(x)$ of your estimate. For the same value of λ, also plot the function $\sqrt{\mathrm{Var}[\widehat{u}(x)]}$ using equation (18.6.8). (You need an estimate of your measurement covariance matrix for this.)

As you change λ you will see very explicitly the trade-off between resolution and stability. Pick the value that meets your needs. You can even choose λ to be a function of x, $\lambda = \lambda(x)$, in equations (18.6.12) and (18.6.13), should you desire to do so. (This is one benefit of solving a separate set of equations for each x.) For the chosen value or values of λ, you now have a quantitative understanding of your inverse solution procedure. This can prove invaluable if — once you are processing real data — you need to judge whether a particular feature, a spike or jump for example, is genuine, and/or is actually resolved. The Backus-Gilbert method has found particular success among geophysicists, who use it to obtain information about the structure of the Earth (e.g., density run with depth) from seismic travel time data.

CITED REFERENCES AND FURTHER READING:

Backus, G.E., and Gilbert, F. 1968, *Geophysical Journal of the Royal Astronomical Society*, vol. 16, pp. 169–205. [1]

Backus, G.E., and Gilbert, F. 1970, *Philosophical Transactions of the Royal Society of London A*, vol. 266, pp. 123–192. [2]

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64. [3]

Loredo, T.J., and Epstein, R.I. 1989, *Astrophysical Journal*, vol. 336, pp. 896–919. [4]

18.7 Maximum Entropy Image Restoration

Above, we commented that the association of certain inversion methods with Bayesian arguments is more historical accident than intellectual imperative. *Maximum entropy methods*, so-called, are notorious in this regard; to summarize these methods without some, at least introductory, Bayesian invocations would be to serve a steak without the sizzle, or a sundae without the cherry. We should

also comment in passing that the connection between maximum entropy inversion methods, considered here, and maximum entropy spectral estimation, discussed in §13.7, is rather abstract. For practical purposes the two techniques, though both named *maximum entropy method* or *MEM*, are unrelated.

Bayes' Theorem, which follows from the standard axioms of probability, relates the conditional probabilities of two events, say A and B:

$$\text{Prob}(A|B) = \text{Prob}(A)\,\frac{\text{Prob}(B|A)}{\text{Prob}(B)} \tag{18.7.1}$$

Here $\text{Prob}(A|B)$ is the probability of A *given* that B has occurred, and similarly for $\text{Prob}(B|A)$, while $\text{Prob}(A)$ and $\text{Prob}(B)$ are unconditional probabilities.

"Bayesians" (so-called) adopt a broader interpretation of probabilities than do so-called "frequentists." To a Bayesian, $P(A|B)$ is a measure of the degree of plausibility of A (given B) on a scale ranging from zero to one. In this broader view, A and B need not be repeatable events; they can be propositions or hypotheses. The equations of probability theory then become a set of consistent rules for conducting inference [1,2]. Since plausibility is itself always conditioned on some, perhaps unarticulated, set of assumptions, all Bayesian probabilities are viewed as conditional on some collective background information I.

Suppose H is some hypothesis. Even before there exist any explicit data, a Bayesian can assign to H some degree of plausibility $\text{Prob}(H|I)$, called the "Bayesian prior." Now, when some data D_1 comes along, Bayes theorem tells how to reassess the plausibility of H,

$$\text{Prob}(H|D_1 I) = \text{Prob}(H|I)\frac{\text{Prob}(D_1|HI)}{\text{Prob}(D_1|I)} \tag{18.7.2}$$

The factor in the numerator on the right of equation (18.7.2) is calculable as the probability of a data set *given* the hypothesis (compare with "likelihood" in §15.1). The denominator, called the "prior predictive probability" of the data, is in this case merely a normalization constant which can be calculated by the requirement that the probability of all hypotheses should sum to unity. (In other Bayesian contexts, the prior predictive probabilities of two qualitatively different models can be used to assess their relative plausibility.)

If some additional data D_2 comes along tomorrow, we can further refine our estimate of H's probability, as

$$\text{Prob}(H|D_2 D_1 I) = \text{Prob}(H|D_1 I)\frac{\text{Prob}(D_2|HD_1 I)}{\text{Prob}(D_1|D_1 I)} \tag{18.7.3}$$

Using the product rule for probabilities, $\text{Prob}(AB|C) = \text{Prob}(A|C)\text{Prob}(B|AC)$, we find that equations (18.7.2) and (18.7.3) imply

$$\text{Prob}(H|D_2 D_1 I) = \text{Prob}(H|I)\frac{\text{Prob}(D_2 D_1|HI)}{\text{Prob}(D_2 D_1|I)} \tag{18.7.4}$$

which shows that we would have gotten the same answer if all the data $D_1 D_2$ had been taken together.

From a Bayesian perspective, inverse problems are inference problems [3,4]. The underlying parameter set **u** is a hypothesis whose probability, given the measured data values **c**, and the Bayesian prior Prob($\mathbf{u}|I$) can be calculated. We might want to report a single "best" inverse **u**, the one that maximizes

$$\text{Prob}(\mathbf{u}|\mathbf{c}I) = \text{Prob}(\mathbf{c}|\mathbf{u}I)\frac{\text{Prob}(\mathbf{u}|I)}{\text{Prob}(\mathbf{c}|I)} \tag{18.7.5}$$

over all possible choices of **u**. Bayesian analysis also admits the possibility of reporting additional information that characterizes the region of possible **u**'s with high relative probability, the so-called "posterior bubble" in **u**.

The calculation of the probability of the data **c**, given the hypothesis **u** proceeds exactly as in the maximum likelihood method. For Gaussian errors, e.g., it is given by

$$\text{Prob}(\mathbf{c}|\mathbf{u}I) = \exp(-\frac{1}{2}\chi^2)\Delta u_1 \Delta u_2 \cdots \Delta u_M \tag{18.7.6}$$

where χ^2 is calculated from **u** and **c** using equation (18.4.9), and the Δu_μ's are constant, small ranges of the components of **u** whose actual magnitude is irrelevant, because they do not depend on **u** (compare equations 15.1.3 and 15.1.4).

In maximum likelihood estimation we, in effect, chose the prior Prob($\mathbf{u}|I$) to be constant. That was a luxury that we could afford when estimating a small number of parameters from a large amount of data. Here, the number of "parameters" (components of **u**) is comparable to or larger than the number of measured values (components of **c**); we *need* to have a nontrivial prior, Prob($\mathbf{u}|I$), to resolve the degeneracy of the solution.

In maximum entropy image restoration, that is where *entropy* comes in. The entropy of a physical system in some macroscopic state, usually denoted S, is the logarithm of the number of microscopically distinct configurations that all have the same macroscopic observables (i.e., consistent with the observed macroscopic state). Actually, we will find it useful to denote the *negative* of the entropy, also called the *negentropy*, by $H \equiv -S$ (a notation that goes back to Boltzmann). In situations where there is reason to believe that the *a priori* probabilities of the *microscopic* configurations are all the same (these situations are called *ergodic*), then the Bayesian prior Prob($\mathbf{u}|I$) for a *macroscopic* state with entropy S is proportional to $\exp(S)$ or $\exp(-H)$.

MEM uses this concept to assign a prior probability to any given underlying function **u**. For example [5-7], suppose that the measurement of luminance in each pixel is quantized to (in some units) an integer value. Let

$$U = \sum_{\mu=1}^{M} u_\mu \tag{18.7.7}$$

be the total number of luminance quanta in the whole image. Then we can base our "prior" on the notion that each luminance quantum has an equal *a priori* chance of being in any pixel. (See [8] for a more abstract justification of this idea.) The number of ways of getting a particular configuration **u** is

$$\frac{U!}{u_1!u_2!\cdots u_M!} \propto \exp\left[-\sum_\mu u_\mu \ln(u_\mu/U) + \frac{1}{2}\left(\ln U - \sum_\mu \ln u_\mu\right)\right] \tag{18.7.8}$$

Here the left side can be understood as the number of distinct orderings of all the luminance quanta, divided by the numbers of equivalent reorderings within each pixel, while the right side follows by Stirling's approximation to the factorial function. Taking the negative of the logarithm, and neglecting terms of order $\log U$ in the presence of terms of order U, we get the negentropy

$$H(\mathbf{u}) = \sum_{\mu=1}^{M} u_\mu \ln(u_\mu/U) \tag{18.7.9}$$

From equations (18.7.5), (18.7.6), and (18.7.9) we now seek to maximize

$$\text{Prob}(\mathbf{u}|\mathbf{c}) \propto \exp\left[-\frac{1}{2}\chi^2\right] \exp[-H(\mathbf{u})] \tag{18.7.10}$$

or, equivalently,

$$\text{minimize:} \quad -\ln[\,\text{Prob}(\mathbf{u}|\mathbf{c})\,] = \frac{1}{2}\chi^2[\mathbf{u}] + H(\mathbf{u}) = \frac{1}{2}\chi^2[\mathbf{u}] + \sum_{\mu=1}^{M} u_\mu \ln(u_\mu/U) \tag{18.7.11}$$

This ought to remind you of equation (18.4.11), or equation (18.5.6), or in fact any of our previous minimization principles along the lines of $\mathcal{A} + \lambda\mathcal{B}$, where $\lambda\mathcal{B} = H(\mathbf{u})$ is a regularizing operator. Where is λ? We need to put it in for exactly the reason discussed following equation (18.4.11): Degenerate inversions are likely to be able to achieve unrealistically small values of χ^2. We need an adjustable parameter to bring χ^2 into its expected narrow statistical range of $N \pm (2N)^{1/2}$. The discussion at the beginning of §18.4 showed that it makes no difference which term we attach the λ to. For consistency in notation, we absorb a factor 2 into λ and put it on the entropy term. (Another way to see the necessity of an undetermined λ factor is to note that it is necessary if our minimization principle is to be invariant under changing the units in which $\mathbf{u}$ is quantized, e.g., if an 8-bit analog-to-digital converter is replaced by a 12-bit one.) We can now also put "hats" back to indicate that this is the procedure for obtaining our chosen statistical estimator:

$$\text{minimize:} \quad \mathcal{A} + \lambda\mathcal{B} = \chi^2[\widehat{\mathbf{u}}] + \lambda H(\widehat{\mathbf{u}}) = \chi^2[\widehat{\mathbf{u}}] + \lambda \sum_{\mu=1}^{M} \widehat{u}_\mu \ln(\widehat{u}_\mu) \tag{18.7.12}$$

(Formally, we might also add a second Lagrange multiplier $\lambda' U$, to constrain the total intensity U to be constant.)

It is not hard to see that the negentropy, $H(\widehat{\mathbf{u}})$, is in fact a regularizing operator, similar to $\widehat{\mathbf{u}} \cdot \widehat{\mathbf{u}}$ (equation 18.4.11) or $\widehat{\mathbf{u}} \cdot \mathbf{H} \cdot \widehat{\mathbf{u}}$ (equation 18.5.6). The following of its properties are noteworthy:

1. When U is held constant, $H(\widehat{\mathbf{u}})$ is minimized for $\widehat{u}_\mu = U/M =$ constant, so it smooths in the sense of trying to achieve a constant solution, similar to equation (18.5.4). The fact that the constant solution is a minimum follows from the fact that the second derivative of $u \ln u$ is positive.

2. Unlike equation (18.5.4), however, $H(\hat{\mathbf{u}})$ is *local*, in the sense that it does not difference neighboring pixels. It simply sums some function f, here

$$f(u) = u \ln u \tag{18.7.13}$$

over all pixels; it is invariant, in fact, under a complete scrambling of the pixels in an image. This form implies that $H(\hat{\mathbf{u}})$ is not seriously increased by the occurrence of a small number of very bright pixels (point sources) embedded in a low-intensity smooth background.
3. $H(\hat{\mathbf{u}})$ goes to infinite slope as any one pixel goes to zero. This causes it to enforce positivity of the image, without the necessity of additional deterministic constraints.
4. The biggest difference between $H(\hat{\mathbf{u}})$ and the other regularizing operators that we have met is that $H(\hat{\mathbf{u}})$ is not a quadratic functional of $\hat{\mathbf{u}}$, so the equations obtained by varying equation (18.7.12) are *nonlinear*. This fact is itself worthy of some additional discussion.

Nonlinear equations are harder to solve than linear equations. For image processing, however, the large number of equations usually dictates an iterative solution procedure, even for linear equations, so the practical effect of the nonlinearity is somewhat mitigated. Below, we will summarize some of the methods that are successfully used for MEM inverse problems.

For some problems, notably the problem in radio-astronomy of image recovery from an incomplete set of Fourier coefficients, the superior performance of MEM inversion can be, in part, traced to the nonlinearity of $H(\hat{\mathbf{u}})$. One way to see this [5] is to consider the limit of perfect measurements $\sigma_i \to 0$. In this case the χ^2 term in the minimization principle (18.7.12) gets replaced by a set of constraints, each with its own Lagrange multiplier, requiring agreement between model and data; that is,

$$\text{minimize:} \quad \sum_j \lambda_j \left[c_j - \sum_\mu R_{j\mu} \hat{u}_\mu \right] + H(\hat{\mathbf{u}}) \tag{18.7.14}$$

(cf. equation 18.4.7). Setting the formal derivative with respect to $\hat{u}_\mu$ to zero gives

$$\frac{\partial H}{\partial \hat{u}_\mu} = f'(\hat{u}_\mu) = \sum_j \lambda_j R_{j\mu} \tag{18.7.15}$$

or defining a function G as the inverse function of f',

$$\hat{u}_\mu = G\left(\sum_j \lambda_j R_{j\mu} \right) \tag{18.7.16}$$

This solution is only formal, since the λ_j's must be found by requiring that equation (18.7.16) satisfy all the constraints built into equation (18.7.14). However, equation (18.7.16) does show the crucial fact that if G is *linear*, then the solution $\hat{\mathbf{u}}$ contains *only* a linear combination of basis functions $R_{j\mu}$ corresponding to actual measurements j. This is equivalent to setting unmeasured c_j's to zero. Notice that the principal solution obtained from equation (18.4.11) in fact has a linear G.

In the problem of incomplete Fourier image reconstruction, the typical $R_{j\mu}$ has the form $\exp(-2\pi i \mathbf{k}_j \cdot \mathbf{x}_\mu)$, where $\mathbf{x}_\mu$ is a two-dimensional vector in the image space and $\mathbf{k}_\mu$ is a two-dimensional wave-vector. If an image contains strong point sources, then the effect of setting unmeasured c_j's to zero is to produce sidelobe ripples throughout the image plane. These ripples can mask any actual extended, low-intensity image features lying between the point sources. If, however, the slope of G is smaller for small values of its argument, larger for large values, then ripples in low-intensity portions of the image are relatively suppressed, while strong point sources will be relatively sharpened ("superresolution"). This behavior on the slope of G is equivalent to requiring $f'''(u) < 0$. For $f(u) = u \ln u$, we in fact have $f'''(u) = -1/u^2 < 0$.

In more picturesque language, the nonlinearity acts to "create" nonzero values for the unmeasured c_i's, so as to suppress the low-intensity ripple and sharpen the point sources.

Is MEM Really Magical?

How unique is the negentropy functional (18.7.9)? Recall that that equation is based on the assumption that luminance elements are *a priori* distributed over the pixels uniformly. If we instead had some other preferred *a priori* image in mind, one with pixel intensities m_μ, then it is easy to show that the negentropy becomes

$$H(\mathbf{u}) = \sum_{\mu=1}^{M} u_\mu \ln(u_\mu/m_\mu) + \text{constant} \tag{18.7.17}$$

(the constant can then be ignored). All the rest of the discussion then goes through.

More fundamentally, and despite statements by zealots to the contrary [7], there is actually nothing universal about the functional form $f(u) = u \ln u$. In some other physical situations (for example, the entropy of an electromagnetic field in the limit of many photons per mode, as in radio-astronomy) the physical negentropy functional is actually $f(u) = -\ln u$ (see [5] for other examples). In general, the question, "Entropy of what?" is not uniquely answerable in any particular situation. (See reference [9] for an attempt at articulating a more general principle that reduces to one or another entropy functional under appropriate circumstances.)

The four numbered properties summarized above, plus the desirable sign for nonlinearity, $f'''(u) < 0$, are all as true for $f(u) = -\ln u$ as for $f(u) = u \ln u$. In fact these properties are shared by a nonlinear function as simple as $f(u) = -\sqrt{u}$, which has no information theoretic justification at all (no logarithms!). MEM reconstructions of test images using any of these entropy forms are virtually indistinguishable [5].

By all available evidence, MEM seems to be neither more nor less than one usefully nonlinear version of the general regularization scheme $\mathcal{A} + \lambda\mathcal{B}$ that we have by now considered in many forms. Its peculiarities become strengths when applied to the reconstruction from incomplete Fourier data of images that are expected to be dominated by very bright point sources, but which also contain interesting low-intensity, extended sources. For images of some other character, there is no reason to suppose that MEM methods will generally dominate other regularization schemes, either ones already known or yet to be invented.

Algorithms for MEM

The goal is to find the vector $\widehat{\mathbf{u}}$ that minimizes $\mathcal{A} + \lambda\mathcal{B}$ where in the notation of equations (18.5.5), (18.5.6), and (18.7.13),

$$\mathcal{A} = |\mathbf{b} - \mathbf{A} \cdot \widehat{\mathbf{u}}|^2 \qquad \mathcal{B} = \sum_\mu f(\widehat{u}_\mu) \tag{18.7.18}$$

Compared with a "general" minimization problem, we have the advantage that we can compute the gradients and the second partial derivative matrices (Hessian matrices) explicitly,

$$\begin{aligned} \nabla\mathcal{A} &= 2(\mathbf{A}^T \cdot \mathbf{A} \cdot \widehat{\mathbf{u}} - \mathbf{A}^T \cdot \mathbf{b}) & \frac{\partial^2\mathcal{A}}{\partial\widehat{u}_\mu \partial\widehat{u}_\rho} &= [2\mathbf{A}^T \cdot \mathbf{A}]_{\mu\rho} \\ [\nabla\mathcal{B}]_\mu &= f'(\widehat{u}_\mu) & \frac{\partial^2\mathcal{B}}{\partial\widehat{u}_\mu \partial\widehat{u}_\rho} &= \delta_{\mu\rho} f''(\widehat{u}_\mu) \end{aligned} \tag{18.7.19}$$

It is important to note that while $\mathcal{A}$'s second partial derivative matrix cannot be stored (its size is the square of the number of pixels), it can be applied to any vector by first applying $\mathbf{A}$, then $\mathbf{A}^T$. In the case of reconstruction from incomplete Fourier data, or in the case of convolution with a translation invariant point spread function, these applications will typically involve several FFTs. Likewise, the calculation of the gradient $\nabla\mathcal{A}$ will involve FFTs in the application of $\mathbf{A}$ and $\mathbf{A}^T$.

While some success has been achieved with the classical conjugate gradient method (§10.6), it is often found that the nonlinearity in $f(u) = u \ln u$ causes problems. Attempted steps that give $\widehat{\mathbf{u}}$ with even one negative value must be cut in magnitude, sometimes so severely as to slow the solution to a crawl. The underlying problem is that the conjugate gradient method develops its information about the inverse of the Hessian matrix a bit at a time, while changing its location in the search space. When a nonlinear function is quite different from a pure quadratic form, the old information becomes obsolete before it gets usefully exploited.

Skilling and collaborators [6,7,10,11] developed a complicated but highly successful scheme, wherein a minimum is repeatedly sought not along a single search direction, but in a small- (typically three-) dimensional subspace, spanned by vectors that are calculated anew at each landing point. The subspace basis vectors are chosen in such a way as to avoid directions leading to negative values. One of the most successful choices is the three-dimensional subspace spanned by the vectors with components given by

$$\begin{aligned} e_\mu^{(1)} &= \widehat{u}_\mu [\nabla\mathcal{A}]_\mu \\ e_\mu^{(2)} &= \widehat{u}_\mu [\nabla\mathcal{B}]_\mu \\ e_\mu^{(3)} &= \frac{\widehat{u}_\mu \sum_\rho (\partial^2\mathcal{A}/\partial\widehat{u}_\mu\partial\widehat{u}_\rho)\widehat{u}_\rho [\nabla\mathcal{B}]_\rho}{\sqrt{\sum_\rho \widehat{u}_\rho \left([\nabla\mathcal{B}]_\rho\right)^2}} - \frac{\widehat{u}_\mu \sum_\rho (\partial^2\mathcal{A}/\partial\widehat{u}_\mu\partial\widehat{u}_\rho)\widehat{u}_\rho [\nabla\mathcal{A}]_\rho}{\sqrt{\sum_\rho \widehat{u}_\rho \left([\nabla\mathcal{A}]_\rho\right)^2}} \end{aligned} \tag{18.7.20}$$

(In these equations there is no sum over μ.) The form of the $\mathbf{e}^{(3)}$ has some justification if one views dot products as occurring in a space with the metric $g_{\mu\nu} = \delta_{\mu\nu}/u_\mu$, chosen to make zero values "far away"; see [6].

Within the three-dimensional subspace, the three-component gradient and nine-component Hessian matrix are computed by projection from the large space, and the minimum in the subspace is estimated by (trivially) solving three simultaneous linear equations, as in §10.7, equation (10.7.4). The size of a step $\Delta\widehat{\mathbf{u}}$ is required to be limited by the inequality

$$\sum_{\mu}(\Delta\widehat{u}_\mu)^2/\widehat{u}_\mu < (0.1 \text{ to } 0.5)U \tag{18.7.21}$$

Because the gradient directions $\nabla\mathcal{A}$ and $\nabla\mathcal{B}$ are separately available, it is possible to combine the minimum search with a simultaneous adjustment of λ so as finally to satisfy the desired constraint. There are various further tricks employed.

A less general, but in practice often equally satisfactory, approach is due to Cornwell and Evans [12]. Here, noting that $\mathcal{B}$'s Hessian (second partial derivative) matrix is diagonal, one asks whether there is a useful diagonal approximation to $\mathcal{A}$'s Hessian, namely $2\mathbf{A}^T \cdot \mathbf{A}$. If Λ_μ denotes the diagonal components of such an approximation, then a useful step in $\widehat{\mathbf{u}}$ would be

$$\Delta\widehat{u}_\mu = -\frac{1}{\Lambda_\mu + \lambda f''(\widehat{u}_\mu)}(\nabla\mathcal{A} + \lambda\nabla\mathcal{B}) \tag{18.7.22}$$

(again compare equation 10.7.4). Even more extreme, one might seek an approximation with constant diagonal elements, $\Lambda_\mu = \Lambda$, so that

$$\Delta\widehat{u}_\mu = -\frac{1}{\Lambda + \lambda f''(\widehat{u}_\mu)}(\nabla\mathcal{A} + \lambda\nabla\mathcal{B}) \tag{18.7.23}$$

Since $\mathbf{A}^T \cdot \mathbf{A}$ has something of the nature of a doubly convolved point spread function, and since in real cases one often has a point spread function with a sharp central peak, even the more extreme of these approximations is often fruitful. One starts with a rough estimate of Λ obtained from the $A_{i\mu}$'s, e.g.,

$$\Lambda \sim \left\langle \sum_i [A_{i\mu}]^2 \right\rangle \tag{18.7.24}$$

An accurate value is not important, since in practice Λ is adjusted adaptively: If Λ is too large, then equation (18.7.23)'s steps will be too small (that is, larger steps in the same direction will produce even greater decrease in $\mathcal{A} + \lambda\mathcal{B}$). If Λ is too small, then attempted steps will land in an unfeasible region (negative values of $\widehat{u}_\mu$), or will result in an increased $\mathcal{A} + \lambda\mathcal{B}$. There is an obvious similarity between the adjustment of Λ here and the Levenberg-Marquardt method of §15.5; this should not be too surprising, since MEM is closely akin to the problem of nonlinear least-squares fitting. Reference [12] also discusses how the value of $\Lambda + \lambda f''(\widehat{u}_\mu)$ can be used to adjust the Lagrange multiplier λ so as to converge to the desired value of χ^2.

All practical MEM algorithms are found to require on the order of 30 to 50 iterations to converge. This convergence behavior is not now understood in any fundamental way.

"Bayesian" versus "Historic" Maximum Entropy

Several more recent developments in maximum entropy image restoration go under the rubric "Bayesian" to distinguish them from the previous "historic" methods. See [13] for details and references.

- Better priors: We already noted that the entropy functional (equation 18.7.13) is invariant under scrambling all pixels and has no notion of smoothness. The so-called "intrinsic correlation function" (ICF) model (Ref. [13], where it is called "New MaxEnt") is similar enough to the entropy functional to allow similar algorithms, but it makes the values of neighboring pixels correlated, enforcing smoothness.
- Better estimation of λ: Above we chose λ to bring χ^2 into its expected narrow statistical range of $N \pm (2N)^{1/2}$. This in effect overestimates χ^2, however, since some effective number γ of parameters are being "fitted" in doing the reconstruction. A Bayesian approach leads to a self-consistent estimate of this γ and an objectively better choice for λ.

CITED REFERENCES AND FURTHER READING:

Jaynes, E.T. 1976, in *Foundations of Probability Theory, Statistical Inference, and Statistical Theories of Science*, W.L. Harper and C.A. Hooker, eds. (Dordrecht: Reidel). [1]

Jaynes, E.T. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel). [2]

Jaynes, E.T. 1984, in *SIAM-AMS Proceedings*, vol. 14, D.W. McLaughlin, ed. (Providence, RI: American Mathematical Society). [3]

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, 381–387. [4]

Narayan, R., and Nityananda, R. 1986, *Annual Review of Astronomy and Astrophysics*, vol. 24, pp. 127–170. [5]

Skilling, J., and Bryan, R.K. 1984, *Monthly Notices of the Royal Astronomical Society*, vol. 211, pp. 111–124. [6]

Burch, S.F., Gull, S.F., and Skilling, J. 1983, *Computer Vision, Graphics and Image Processing*, vol. 23, pp. 113–128. [7]

Skilling, J. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [8]

Frieden, B.R. 1983, *Journal of the Optical Society of America*, vol. 73, pp. 927–938. [9]

Skilling, J., and Gull, S.F. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel). [10]

Skilling, J. 1986, in *Maximum Entropy and Bayesian Methods in Applied Statistics*, J.H. Justice, ed. (Cambridge: Cambridge University Press). [11]

Cornwell, T.J., and Evans, K.F. 1985, *Astronomy and Astrophysics*, vol. 143, pp. 77–83. [12]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [13]

Chapter 19. Partial Differential Equations

19.0 Introduction

The numerical treatment of partial differential equations is, by itself, a vast subject. Partial differential equations are at the heart of many, if not most, computer analyses or simulations of continuous physical systems, such as fluids, electromagnetic fields, the human body, and so on. The intent of this chapter is to give the briefest possible useful introduction. Ideally, there would be an entire second volume of *Numerical Recipes* dealing with partial differential equations alone. (The references [1-4] provide, of course, available alternatives.)

In most mathematics books, partial differential equations (PDEs) are classified into the three categories, *hyperbolic*, *parabolic*, and *elliptic*, on the basis of their *characteristics*, or curves of information propagation. The prototypical example of a hyperbolic equation is the one-dimensional *wave* equation

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.0.1}$$

where $v =$ constant is the velocity of wave propagation. The prototypical parabolic equation is the *diffusion* equation

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D \frac{\partial u}{\partial x}\right) \tag{19.0.2}$$

where D is the diffusion coefficient. The prototypical elliptic equation is the *Poisson* equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x, y) \tag{19.0.3}$$

where the source term ρ is given. If the source term is equal to zero, the equation is *Laplace's equation*.

From a computational point of view, the classification into these three canonical types is not very meaningful — or at least not as important as some other essential distinctions. Equations (19.0.1) and (19.0.2) both define *initial value* or *Cauchy* problems: If information on u (perhaps including time derivative information) is

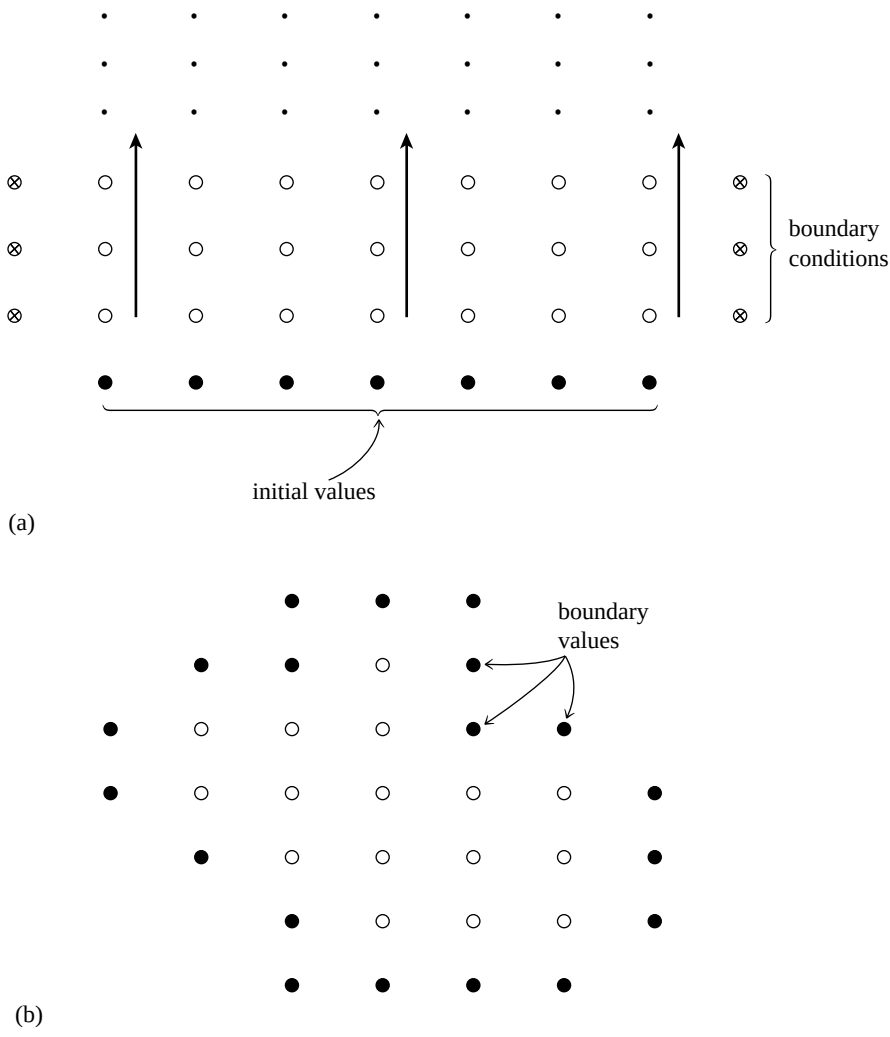


Figure 19.0.1. Initial value problem (a) and boundary value problem (b) are contrasted. In (a) initial values are given on one "time slice," and it is desired to advance the solution in time, computing successive rows of open dots in the direction shown by the arrows. Boundary conditions at the left and right edges of each row ($\otimes$) must also be supplied, but only one row at a time. Only one, or a few, previous rows need be maintained in memory. In (b), boundary values are specified around the edge of a grid, and an iterative process is employed to find the values of all the internal points (open circles). All grid points must be maintained in memory.

given at some initial time t_0 for all x, then the equations describe how $u(x,t)$ propagates itself forward in time. In other words, equations (19.0.1) and (19.0.2) describe time evolution. The goal of a numerical code should be to track that time evolution with some desired accuracy.

By contrast, equation (19.0.3) directs us to find a single "static" function $u(x,y)$ which satisfies the equation within some (x,y) region of interest, and which — one must also specify — has some desired behavior on the boundary of the region of interest. These problems are called *boundary value problems*. In general it is not

possible stably to just "integrate in from the boundary" in the same sense that an initial value problem can be "integrated forward in time." Therefore, the goal of a numerical code is somehow to converge on the correct solution everywhere at once.

This, then, is the most important classification from a computational point of view: Is the problem at hand an *initial value* (time evolution) problem? or is it a *boundary value* (static solution) problem? Figure 19.0.1 emphasizes the distinction. Notice that while the italicized terminology is standard, the terminology in parentheses is a much better description of the dichotomy from a computational perspective. The subclassification of initial value problems into parabolic and hyperbolic is much less important because (i) many actual problems are of a mixed type, and (ii) as we will see, most hyperbolic problems get parabolic pieces mixed into them by the time one is discussing practical computational schemes.

Initial Value Problems

An initial value problem is defined by answers to the following questions:

- What are the dependent variables to be propagated forward in time?
- What is the evolution equation for each variable? Usually the evolution equations will all be coupled, with more than one dependent variable appearing on the right-hand side of each equation.
- What is the highest time derivative that occurs in each variable's evolution equation? If possible, this time derivative should be put alone on the equation's left-hand side. Not only the value of a variable, but also the value of all its time derivatives — up to the highest one — must be specified to define the evolution.
- What special equations (boundary conditions) govern the evolution in time of points on the boundary of the spatial region of interest? Examples: *Dirichlet conditions* specify the values of the boundary points as a function of time; *Neumann conditions* specify the values of the normal gradients on the boundary; *outgoing-wave boundary conditions* are just what they say.

Sections 19.1–19.3 of this chapter deal with initial value problems of several different forms. We make no pretence of completeness, but rather hope to convey a certain amount of generalizable information through a few carefully chosen model examples. These examples will illustrate an important point: One's principal *computational* concern must be the *stability* of the algorithm. Many reasonable-looking algorithms for initial value problems just don't work — they are numerically unstable.

Boundary Value Problems

The questions that define a boundary value problem are:

- What are the variables?
- What equations are satisfied in the interior of the region of interest?
- What equations are satisfied by points on the boundary of the region of interest? (Here Dirichlet and Neumann conditions are possible choices for elliptic second-order equations, but more complicated boundary conditions can also be encountered.)

In contrast to initial value problems, stability is relatively easy to achieve for boundary value problems. Thus, the *efficiency* of the algorithms, both in computational load and storage requirements, becomes the principal concern.

Because all the conditions on a boundary value problem must be satisfied "simultaneously," these problems usually boil down, at least conceptually, to the solution of large numbers of simultaneous algebraic equations. When such equations are nonlinear, they are usually solved by linearization and iteration; so without much loss of generality we can view the problem as being the solution of special, large linear sets of equations.

As an example, one which we will refer to in §§19.4–19.6 as our "model problem," let us consider the solution of equation (19.0.3) by the *finite-difference method*. We represent the function $u(x, y)$ by its values at the discrete set of points

$$\begin{aligned} x_j &= x_0 + j\Delta, \qquad j = 0, 1, ..., J \\ y_l &= y_0 + l\Delta, \qquad l = 0, 1, ..., L \end{aligned} \tag{19.0.4}$$

where Δ is the *grid spacing*. From now on, we will write $u_{j,l}$ for $u(x_j, y_l)$, and $\rho_{j,l}$ for $\rho(x_j, y_l)$. For (19.0.3) we substitute a finite-difference representation (see Figure 19.0.2),

$$\frac{u_{j+1,l} - 2u_{j,l} + u_{j-1,l}}{\Delta^2} + \frac{u_{j,l+1} - 2u_{j,l} + u_{j,l-1}}{\Delta^2} = \rho_{j,l} \tag{19.0.5}$$

or equivalently

$$u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l} = \Delta^2 \rho_{j,l} \tag{19.0.6}$$

To write this system of linear equations in matrix form we need to make a vector out of u. Let us number the two dimensions of grid points in a single one-dimensional sequence by defining

$$i \equiv j(L+1) + l \qquad \text{for} \qquad j = 0, 1, ..., J, \qquad l = 0, 1, ..., L \tag{19.0.7}$$

In other words, i increases most rapidly along the columns representing y values. Equation (19.0.6) now becomes

$$u_{i+L+1} + u_{i-(L+1)} + u_{i+1} + u_{i-1} - 4u_i = \Delta^2 \rho_i \tag{19.0.8}$$

This equation holds only at the interior points $j = 1, 2, ..., J-1; l = 1, 2, ..., L-1$.

The points where

$$\begin{aligned} j &= 0 \qquad [\text{i.e., } i = 0, ..., L] \\ j &= J \qquad [\text{i.e., } i = J(L+1), ..., J(L+1) + L] \\ l &= 0 \qquad [\text{i.e., } i = 0, L+1, ..., J(L+1)] \\ l &= L \qquad [\text{i.e., } i = L, L+1+L, ..., J(L+1) + L] \end{aligned} \tag{19.0.9}$$

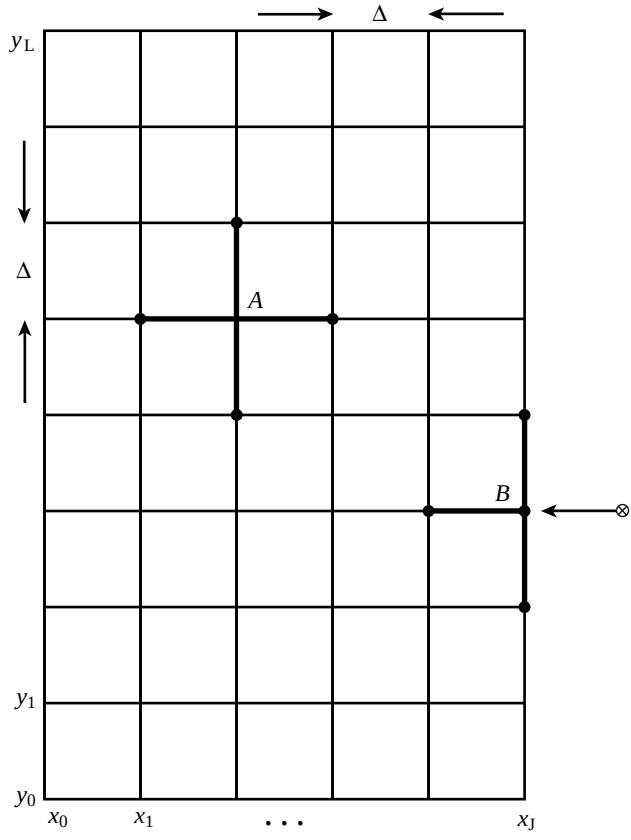


Figure 19.0.2. Finite-difference representation of a second-order elliptic equation on a two-dimensional grid. The second derivatives at the point A are evaluated using the points to which A is shown connected. The second derivatives at point B are evaluated using the connected points and also using "right-hand side" boundary information, shown schematically as $\otimes$.

are boundary points where either u or its derivative has been specified. If we pull all this "known" information over to the right-hand side of equation (19.0.8), then the equation takes the form

$$\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.0.10}$$

where **A** has the form shown in Figure 19.0.3. The matrix **A** is called "tridiagonal with fringes." A general linear second-order elliptic equation

$$\begin{aligned} a(x,y)\frac{\partial^2 u}{\partial x^2} + b(x,y)\frac{\partial u}{\partial x} + c(x,y)\frac{\partial^2 u}{\partial y^2} + d(x,y)\frac{\partial u}{\partial y} \\ + e(x,y)\frac{\partial^2 u}{\partial x \partial y} + f(x,y)u = g(x,y) \end{aligned} \tag{19.0.11}$$

will lead to a matrix of similar structure except that the nonzero entries will not be constants.

As a rough classification, there are three different approaches to the solution of equation (19.0.10), not all applicable in all cases: relaxation methods, "rapid" methods (e.g., Fourier methods), and direct matrix methods.

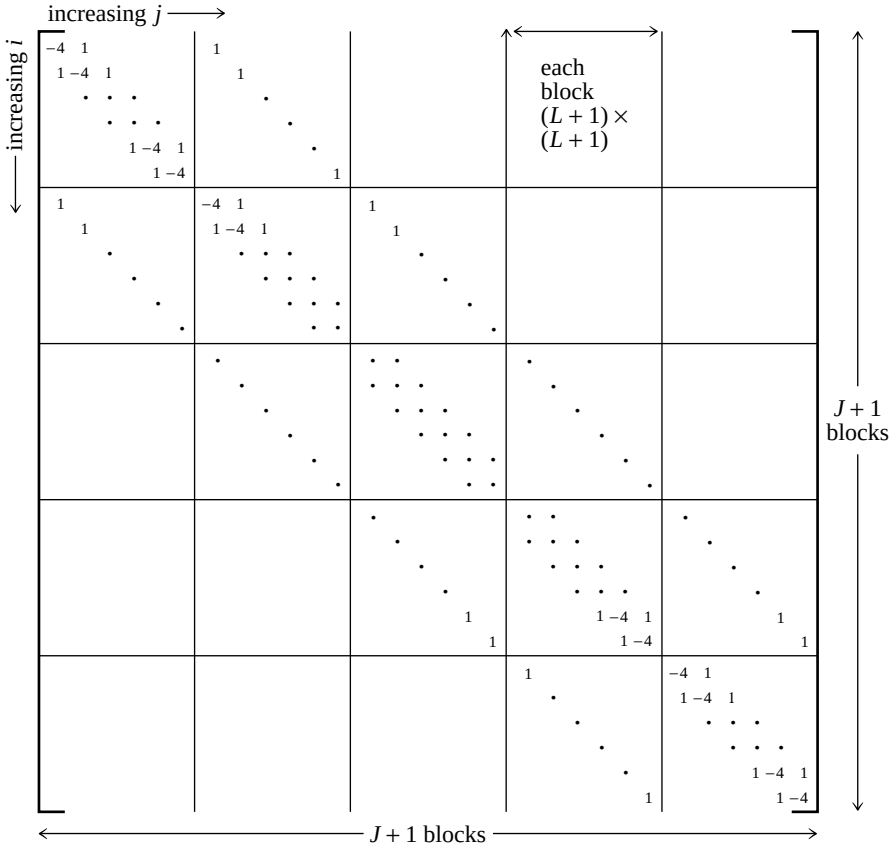


Figure 19.0.3. Matrix structure derived from a second-order elliptic equation (here equation 19.0.6). All elements not shown are zero. The matrix has diagonal blocks that are themselves tridiagonal, and sub- and super-diagonal blocks that are diagonal. This form is called "tridiagonal with fringes." A matrix this sparse would never be stored in its full form as shown here.

Relaxation methods make immediate use of the structure of the sparse matrix **A**. The matrix is split into two parts

$$\mathbf{A} = \mathbf{E} - \mathbf{F} \tag{19.0.12}$$

where **E** is easily invertible and **F** is the remainder. Then (19.0.10) becomes

$$\mathbf{E} \cdot \mathbf{u} = \mathbf{F} \cdot \mathbf{u} + \mathbf{b} \tag{19.0.13}$$

The relaxation method involves choosing an initial guess $\mathbf{u}^{(0)}$ and then solving successively for iterates $\mathbf{u}^{(r)}$ from

$$\mathbf{E} \cdot \mathbf{u}^{(r)} = \mathbf{F} \cdot \mathbf{u}^{(r-1)} + \mathbf{b} \tag{19.0.14}$$

Since **E** is chosen to be easily invertible, each iteration is fast. We will discuss relaxation methods in some detail in §19.5 and §19.6.

So-called rapid methods [5] apply for only a rather special class of equations: those with constant coefficients, or, more generally, those that are separable in the chosen coordinates. In addition, the boundaries must coincide with coordinate lines. This special class of equations is met quite often in practice. We defer detailed discussion to §19.4. Note, however, that the multigrid relaxation methods discussed in §19.6 can be faster than "rapid" methods.

Matrix methods attempt to solve the equation

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{19.0.15}$$

directly. The degree to which this is practical depends very strongly on the exact structure of the matrix **A** for the problem at hand, so our discussion can go no farther than a few remarks and references at this point.

Sparseness of the matrix *must* be the guiding force. Otherwise the matrix problem is prohibitively large. For example, the simplest problem on a 100×100 spatial grid would involve 10000 unknown $u_{j,l}$'s, implying a 10000×10000 matrix **A**, containing 10^8 elements!

As we discussed at the end of §2.7, if **A** is symmetric and positive definite (as it usually is in elliptic problems), the conjugate-gradient algorithm can be used. In practice, rounding error often spoils the effectiveness of the conjugate gradient algorithm for solving finite-difference equations. However, it is useful when incorporated in methods that first rewrite the equations so that **A** is transformed to a matrix $\mathbf{A}'$ that is close to the identity matrix. The quadratic surface defined by the equations then has almost spherical contours, and the conjugate gradient algorithm works very well. In §2.7, in the routine `linbcg`, an analogous *preconditioner* was exploited for non-positive definite problems with the more general biconjugate gradient method. For the positive definite case that arises in PDEs, an example of a successful implementation is the *incomplete Cholesky conjugate gradient method (ICCG)* (see [6-8]).

Another method that relies on a transformation approach is the *strongly implicit procedure* of Stone [9]. A program called SIPSOL that implements this routine has been published [10].

A third class of matrix methods is the Analyze-Factorize-Operate approach as described in §2.7.

Generally speaking, when you have the storage available to implement these methods — not nearly as much as the 10^8 above, but usually much more than is required by relaxation methods — then you should consider doing so. Only multigrid relaxation methods (§19.6) are competitive with the best matrix methods. For grids larger than, say, 300×300, however, it is generally found that only relaxation methods, or "rapid" methods when they are applicable, are possible.

There Is More to Life than Finite Differencing

Besides finite differencing, there are other methods for solving PDEs. Most important are finite element, Monte Carlo, spectral, and variational methods. Unfortunately, we shall barely be able to do justice to finite differencing in this chapter, and so shall not be able to discuss these other methods in this book. Finite element methods [11-12] are often preferred by practitioners in solid mechanics and structural

engineering; these methods allow considerable freedom in putting computational elements where you want them, important when dealing with highly irregular geometries. Spectral methods [13-15] are preferred for very regular geometries and smooth functions; they converge more rapidly than finite-difference methods (cf. §19.4), but they do not work well for problems with discontinuities.

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press). [1]

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [2]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [3]

Mitchell, A.R., and Griffiths, D.F. 1980, *The Finite Difference Method in Partial Differential Equations* (New York: Wiley) [includes discussion of finite element methods]. [4]

Dorr, F.W. 1970, *SIAM Review*, vol. 12, pp. 248–263. [5]

Meijerink, J.A., and van der Vorst, H.A. 1977, *Mathematics of Computation*, vol. 31, pp. 148–162. [6]

van der Vorst, H.A. 1981, *Journal of Computational Physics*, vol. 44, pp. 1–19 [review of sparse iterative methods]. [7]

Kershaw, D.S. 1970, *Journal of Computational Physics*, vol. 26, pp. 43–65. [8]

Stone, H.J. 1968, *SIAM Journal on Numerical Analysis*, vol. 5, pp. 530–558. [9]

Jesshope, C.R. 1979, *Computer Physics Communications*, vol. 17, pp. 383–391. [10]

Strang, G., and Fix, G. 1973, *An Analysis of the Finite Element Method* (Englewood Cliffs, NJ: Prentice-Hall). [11]

Burnett, D.S. 1987, *Finite Element Analysis: From Concepts to Applications* (Reading, MA: Addison-Wesley). [12]

Gottlieb, D. and Orszag, S.A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphia: S.I.A.M.). [13]

Canuto, C., Hussaini, M.Y., Quarteroni, A., and Zang, T.A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag). [14]

Boyd, J.P. 1989, *Chebyshev and Fourier Spectral Methods* (New York: Springer-Verlag). [15]

19.1 Flux-Conservative Initial Value Problems

A large class of initial value (time-evolution) PDEs in one space dimension can be cast into the form of a *flux-conservative equation*,

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{u})}{\partial x} \tag{19.1.1}$$

where $\mathbf{u}$ and $\mathbf{F}$ are vectors, and where (in some cases) $\mathbf{F}$ may depend not only on $\mathbf{u}$ but also on spatial derivatives of $\mathbf{u}$. The vector $\mathbf{F}$ is called the *conserved flux*.

For example, the prototypical hyperbolic equation, the one-dimensional wave equation with constant velocity of propagation v

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.1.2}$$

engineering; these methods allow considerable freedom in putting computational elements where you want them, important when dealing with highly irregular geometries. Spectral methods [13-15] are preferred for very regular geometries and smooth functions; they converge more rapidly than finite-difference methods (cf. §19.4), but they do not work well for problems with discontinuities.

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press). [1]

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [2]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [3]

Mitchell, A.R., and Griffiths, D.F. 1980, *The Finite Difference Method in Partial Differential Equations* (New York: Wiley) [includes discussion of finite element methods]. [4]

Dorr, F.W. 1970, *SIAM Review*, vol. 12, pp. 248–263. [5]

Meijerink, J.A., and van der Vorst, H.A. 1977, *Mathematics of Computation*, vol. 31, pp. 148–162. [6]

van der Vorst, H.A. 1981, *Journal of Computational Physics*, vol. 44, pp. 1–19 [review of sparse iterative methods]. [7]

Kershaw, D.S. 1970, *Journal of Computational Physics*, vol. 26, pp. 43–65. [8]

Stone, H.J. 1968, *SIAM Journal on Numerical Analysis*, vol. 5, pp. 530–558. [9]

Jesshope, C.R. 1979, *Computer Physics Communications*, vol. 17, pp. 383–391. [10]

Strang, G., and Fix, G. 1973, *An Analysis of the Finite Element Method* (Englewood Cliffs, NJ: Prentice-Hall). [11]

Burnett, D.S. 1987, *Finite Element Analysis: From Concepts to Applications* (Reading, MA: Addison-Wesley). [12]

Gottlieb, D. and Orszag, S.A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphia: S.I.A.M.). [13]

Canuto, C., Hussaini, M.Y., Quarteroni, A., and Zang, T.A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag). [14]

Boyd, J.P. 1989, *Chebyshev and Fourier Spectral Methods* (New York: Springer-Verlag). [15]

19.1 Flux-Conservative Initial Value Problems

A large class of initial value (time-evolution) PDEs in one space dimension can be cast into the form of a *flux-conservative equation*,

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{u})}{\partial x} \tag{19.1.1}$$

where $\mathbf{u}$ and $\mathbf{F}$ are vectors, and where (in some cases) $\mathbf{F}$ may depend not only on $\mathbf{u}$ but also on spatial derivatives of $\mathbf{u}$. The vector $\mathbf{F}$ is called the *conserved flux*.

For example, the prototypical hyperbolic equation, the one-dimensional wave equation with constant velocity of propagation v

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.1.2}$$

can be rewritten as a set of two first-order equations

$$\frac{\partial r}{\partial t} = v\frac{\partial s}{\partial x}$$
$$\frac{\partial s}{\partial t} = v\frac{\partial r}{\partial x} \tag{19.1.3}$$

where

$$r \equiv v\frac{\partial u}{\partial x}$$
$$s \equiv \frac{\partial u}{\partial t} \tag{19.1.4}$$

In this case r and s become the two components of $\mathbf{u}$, and the flux is given by the linear matrix relation

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 0 & -v \\ -v & 0 \end{pmatrix} \cdot \mathbf{u} \tag{19.1.5}$$

(The physicist-reader may recognize equations (19.1.3) as analogous to Maxwell's equations for one-dimensional propagation of electromagnetic waves.)

We will consider, in this section, a prototypical example of the general flux-conservative equation (19.1.1), namely the equation for a scalar u,

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} \tag{19.1.6}$$

with v a constant. As it happens, we already know analytically that the general solution of this equation is a wave propagating in the positive x-direction,

$$u = f(x - vt) \tag{19.1.7}$$

where f is an arbitrary function. However, the numerical strategies that we develop will be equally applicable to the more general equations represented by (19.1.1). In some contexts, equation (19.1.6) is called an *advective* equation, because the quantity u is transported by a "fluid flow" with a velocity v.

How do we go about finite differencing equation (19.1.6) (or, analogously, 19.1.1)? The straightforward approach is to choose equally spaced points along both the t- and x-axes. Thus denote

$$\begin{aligned} x_j &= x_0 + j\Delta x, \qquad j = 0, 1, \ldots, J \\ t_n &= t_0 + n\Delta t, \qquad n = 0, 1, \ldots, N \end{aligned} \tag{19.1.8}$$

Let u_j^n denote $u(t_n, x_j)$. We have several choices for representing the time derivative term. The obvious way is to set

$$\left.\frac{\partial u}{\partial t}\right|_{j,n} = \frac{u_j^{n+1} - u_j^n}{\Delta t} + O(\Delta t) \tag{19.1.9}$$

This is called *forward Euler* differencing (cf. equation 16.1.1). While forward Euler is only first-order accurate in Δt, it has the advantage that one is able to calculate

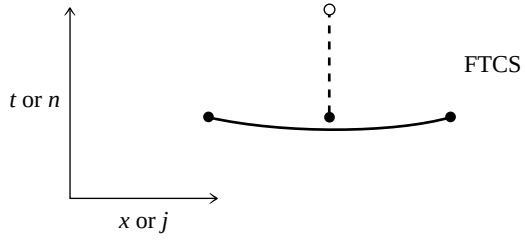


Figure 19.1.1. Representation of the Forward Time Centered Space (FTCS) differencing scheme. In this and subsequent figures, the open circle is the new point at which the solution is desired; filled circles are known points whose function values are used in calculating the new point; the solid lines connect points that are used to calculate spatial derivatives; the dashed lines connect points that are used to calculate time derivatives. The FTCS scheme is generally unstable for hyperbolic problems and cannot usually be used.

quantities at timestep $n+1$ in terms of only quantities known at timestep n. For the space derivative, we can use a second-order representation still using only quantities known at timestep *n*:

$$\left.\frac{\partial u}{\partial x}\right|_{j,n} = \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} + O(\Delta x^2) \tag{19.1.10}$$

The resulting finite-difference approximation to equation (19.1.6) is called the FTCS representation (Forward Time Centered Space),

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) \tag{19.1.11}$$

which can easily be rearranged to be a formula for u_j^{n+1} in terms of the other quantities. The FTCS scheme is illustrated in Figure 19.1.1. It's a fine example of an algorithm that is easy to derive, takes little storage, and executes quickly. Too bad it doesn't work! (See below.)

The FTCS representation is an *explicit* scheme. This means that u_j^{n+1} for each j can be calculated explicitly from the quantities that are already known. Later we shall meet *implicit* schemes, which require us to solve implicit equations coupling the u_j^{n+1} for various j. (Explicit and implicit methods for ordinary differential equations were discussed in §16.6.) The FTCS algorithm is also an example of a *single-level* scheme, since only values at time level n have to be stored to find values at time level $n+1$.

von Neumann Stability Analysis

Unfortunately, equation (19.1.11) is of very limited usefulness. It is an *unstable* method, which can be used only (if at all) to study waves for a short fraction of one oscillation period. To find alternative methods with more general applicability, we must introduce the *von Neumann stability analysis.*

The von Neumann analysis is local: We imagine that the coefficients of the difference equations are so slowly varying as to be considered constant in space and time. In that case, the independent solutions, or *eigenmodes*, of the difference equations are all of the form

$$u_j^n = \xi^n e^{ikj\Delta x} \tag{19.1.12}$$

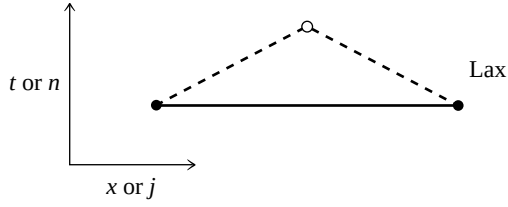


Figure 19.1.2. Representation of the Lax differencing scheme, as in the previous figure. The stability criterion for this scheme is the Courant condition.

where k is a real spatial wave number (which can have any value) and $\xi = \xi(k)$ is a complex number that depends on k. The key fact is that the time dependence of a single eigenmode is nothing more than successive integer powers of the complex number ξ. Therefore, the difference equations are unstable (have exponentially growing modes) if $|\xi(k)| > 1$ for *some* k. The number ξ is called the *amplification factor* at a given wave number k.

To find $\xi(k)$, we simply substitute (19.1.12) back into (19.1.11). Dividing by ξ^n, we get

$$\xi(k) = 1 - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.13}$$

whose modulus is > 1 for *all* k; so the FTCS scheme is unconditionally unstable.

If the velocity v were a function of t and x, then we would write v_j^n in equation (19.1.11). In the von Neumann stability analysis we would still treat v as a constant, the idea being that for v slowly varying the analysis is local. In fact, even in the case of strictly constant v, the von Neumann analysis does not rigorously treat the end effects at $j = 0$ and $j = N$.

More generally, if the equation's right-hand side were nonlinear in u, then a von Neumann analysis would linearize by writing $u = u_0 + \delta u$, expanding to linear order in δu. Assuming that the u_0 quantities already satisfy the difference equation exactly, the analysis would look for an unstable eigenmode of δu.

Despite its lack of rigor, the von Neumann method generally gives valid answers and is much easier to apply than more careful methods. We accordingly adopt it exclusively. (See, for example, [1] for a discussion of other methods of stability analysis.)

Lax Method

The instability in the FTCS method can be cured by a simple change due to Lax. One replaces the term u_j^n in the time derivative term by its average (Figure 19.1.2):

$$u_j^n \to \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) \tag{19.1.14}$$

This turns (19.1.11) into

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) \tag{19.1.15}$$

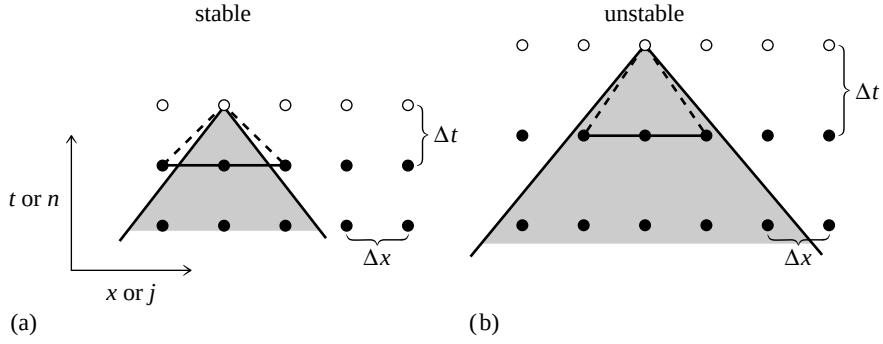


Figure 19.1.3. Courant condition for stability of a differencing scheme. The solution of a hyperbolic problem at a point depends on information within some domain of dependency to the past, shown here shaded. The differencing scheme (19.1.15) has its own domain of dependency determined by the choice of points on one time slice (shown as connected solid dots) whose values are used in determining a new point (shown connected by dashed lines). A differencing scheme is Courant stable if the differencing domain of dependency is larger than that of the PDEs, as in (a), and unstable if the relationship is the reverse, as in (b). For more complicated differencing schemes, the domain of dependency might not be determined simply by the outermost points.

Substituting equation (19.1.12), we find for the amplification factor

$$\xi = \cos k\Delta x - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.16}$$

The stability condition $|\xi|^2 \leq 1$ leads to the requirement

$$\frac{|v|\Delta t}{\Delta x} \leq 1 \tag{19.1.17}$$

This is the famous Courant-Friedrichs-Lewy stability criterion, often called simply the *Courant condition*. Intuitively, the stability condition can be understood as follows (Figure 19.1.3): The quantity u_j^{n+1} in equation (19.1.15) is computed from information at points $j-1$ and $j+1$ at time n. In other words, x_{j-1} and x_{j+1} are the boundaries of the spatial region that is allowed to communicate information to u_j^{n+1}. Now recall that in the continuum wave equation, information actually propagates with a maximum velocity v. If the point u_j^{n+1} is outside of the shaded region in Figure 19.1.3, then it requires information from points more distant than the differencing scheme allows. Lack of that information gives rise to an instability. Therefore, Δt cannot be made too large.

The surprising result, that the simple replacement (19.1.14) stabilizes the FTCS scheme, is our first encounter with the fact that differencing PDEs is an art as much as a science. To see if we can demystify the art somewhat, let us compare the FTCS and Lax schemes by rewriting equation (19.1.15) so that it is in the form of equation (19.1.11) with a remainder term:

$$\frac{u_j^{n+1}-u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) + \frac{1}{2}\left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta t}\right) \tag{19.1.18}$$

But this is exactly the FTCS representation of the equation

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2\Delta t}\nabla^2 u \tag{19.1.19}$$

where $\nabla^2 = \partial^2/\partial x^2$ in one dimension. We have, in effect, added a diffusion term to the equation, or, if you recall the form of the Navier-Stokes equation for viscous fluid flow, a dissipative term. The Lax scheme is thus said to have *numerical dissipation*, or *numerical viscosity*. We can see this also in the amplification factor. Unless $|v|\Delta t$ is exactly equal to Δx, $|\xi| < 1$ and the amplitude of the wave decreases spuriously.

Isn't a spurious decrease as bad as a spurious increase? No. The scales that we hope to study accurately are those that encompass many grid points, so that they have $k\Delta x \ll 1$. (The spatial wave number k is defined by equation 19.1.12.) For these scales, the amplification factor can be seen to be very close to one, in both the stable and unstable schemes. The stable and unstable schemes are therefore about equally accurate. For the unstable scheme, however, short scales with $k\Delta x \sim 1$, *which we are not interested in*, will blow up and swamp the interesting part of the solution. Much better to have a stable scheme in which these short wavelengths die away innocuously. Both the stable and the unstable schemes are *inaccurate* for these short wavelengths, but the inaccuracy is of a tolerable character when the scheme is stable.

When the independent variable **u** is a vector, then the von Neumann analysis is slightly more complicated. For example, we can consider equation (19.1.3), rewritten as

$$\frac{\partial}{\partial t}\begin{bmatrix} r \\ s \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} vs \\ vr \end{bmatrix} \tag{19.1.20}$$

The Lax method for this equation is

$$\begin{aligned} r_j^{n+1} &= \frac{1}{2}(r_{j+1}^n + r_{j-1}^n) + \frac{v\Delta t}{2\Delta x}(s_{j+1}^n - s_{j-1}^n) \\ s_j^{n+1} &= \frac{1}{2}(s_{j+1}^n + s_{j-1}^n) + \frac{v\Delta t}{2\Delta x}(r_{j+1}^n - r_{j-1}^n) \end{aligned} \tag{19.1.21}$$

The von Neumann stability analysis now proceeds by assuming that the eigenmode is of the following (vector) form,

$$\begin{bmatrix} r_j^n \\ s_j^n \end{bmatrix} = \xi^n e^{ikj\Delta x}\begin{bmatrix} r^0 \\ s^0 \end{bmatrix} \tag{19.1.22}$$

Here the vector on the right-hand side is a constant (both in space and in time) eigenvector, and ξ is a complex number, as before. Substituting (19.1.22) into (19.1.21), and dividing by the power ξ^n, gives the homogeneous vector equation

$$\begin{bmatrix} (\cos k\Delta x) - \xi & i\dfrac{v\Delta t}{\Delta x}\sin k\Delta x \\ i\dfrac{v\Delta t}{\Delta x}\sin k\Delta x & (\cos k\Delta x) - \xi \end{bmatrix} \cdot \begin{bmatrix} r^0 \\ s^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{19.1.23}$$

This admits a solution only if the determinant of the matrix on the left vanishes, a condition easily shown to yield the two roots ξ

$$\xi = \cos k\Delta x \pm i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.24}$$

The stability condition is that both roots satisfy $|\xi| \le 1$. This again turns out to be simply the Courant condition (19.1.17).

Other Varieties of Error

Thus far we have been concerned with *amplitude error*, because of its intimate connection with the stability or instability of a differencing scheme. Other varieties of error are relevant when we shift our concern to accuracy, rather than stability.

Finite-difference schemes for hyperbolic equations can exhibit dispersion, or *phase errors*. For example, equation (19.1.16) can be rewritten as

$$\xi = e^{-ik\Delta x} + i\left(1 - \frac{v\Delta t}{\Delta x}\right)\sin k\Delta x \tag{19.1.25}$$

An arbitrary initial wave packet is a superposition of modes with different k's. At each timestep the modes get multiplied by different phase factors (19.1.25), depending on their value of k. If $\Delta t = \Delta x/v$, then the exact solution for each mode of a wave packet $f(x-vt)$ is obtained if each mode gets multiplied by $\exp(-ik\Delta x)$. For this value of Δt, equation (19.1.25) shows that the finite-difference solution gives the exact analytic result. However, if $v\Delta t/\Delta x$ is not exactly 1, the phase relations of the modes can become hopelessly garbled and the wave packet disperses. Note from (19.1.25) that the dispersion becomes large as soon as the wavelength becomes comparable to the grid spacing Δx.

A third type of error is one associated with nonlinear hyperbolic equations and is therefore sometimes called *nonlinear instability*. For example, a piece of the Euler or Navier-Stokes equations for fluid flow looks like

$$\frac{\partial v}{\partial t} = -v\frac{\partial v}{\partial x} + \dots \tag{19.1.26}$$

The nonlinear term in v can cause a transfer of energy in Fourier space from long wavelengths to short wavelengths. This results in a wave profile steepening until a vertical profile or "shock" develops. Since the von Neumann analysis suggests that the stability can depend on $k\Delta x$, a scheme that was stable for shallow profiles can become unstable for steep profiles. This kind of difficulty arises in a differencing scheme where the cascade in Fourier space is halted at the shortest wavelength representable on the grid, that is, at $k \sim 1/\Delta x$. If energy simply accumulates in these modes, it eventually swamps the energy in the long wavelength modes of interest.

Nonlinear instability and shock formation is thus somewhat controlled by numerical viscosity such as that discussed in connection with equation (19.1.18) above. In some fluid problems, however, shock formation is not merely an annoyance, but an actual physical behavior of the fluid whose detailed study is a goal. Then, numerical viscosity alone may not be adequate or sufficiently controllable. This is a complicated subject which we discuss further in the subsection on fluid dynamics, below.

For wave equations, propagation errors (amplitude or phase) are usually most worrisome. For advective equations, on the other hand, *transport errors* are usually of greater concern. In the Lax scheme, equation (19.1.15), a disturbance in the advected quantity u at mesh point j propagates to mesh points $j+1$ and $j-1$ at the next timestep. In reality, however, if the velocity v is positive then only mesh point $j+1$ should be affected.

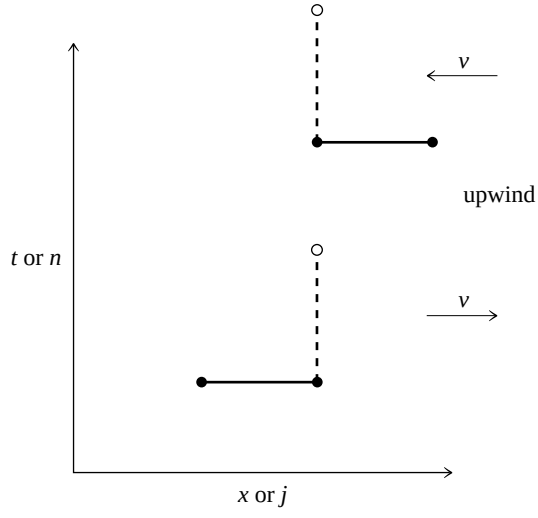


Figure 19.1.4. Representation of upwind differencing schemes. The upper scheme is stable when the advection constant v is negative, as shown; the lower scheme is stable when the advection constant v is positive, also as shown. The Courant condition must, of course, also be satisfied.

The simplest way to model the transport properties "better" is to use *upwind differencing* (see Figure 19.1.4):

$$\frac{u_j^{n+1}-u_j^n}{\Delta t} = -v_j^n \begin{cases} \dfrac{u_j^n - u_{j-1}^n}{\Delta x}, & v_j^n > 0 \\ \dfrac{u_{j+1}^n - u_j^n}{\Delta x}, & v_j^n < 0 \end{cases} \tag{19.1.27}$$

Note that this scheme is only first-order, not second-order, accurate in the calculation of the spatial derivatives. How can it be "better"? The answer is one that annoys the mathematicians: The goal of numerical simulations is not always "accuracy" in a strictly mathematical sense, but sometimes "fidelity" to the underlying physics in a sense that is looser and more pragmatic. In such contexts, some kinds of error are much more tolerable than others. Upwind differencing generally adds fidelity to problems where the advected variables are liable to undergo sudden changes of state, e.g., as they pass through shocks or other discontinuities. You will have to be guided by the specific nature of your own problem.

For the differencing scheme (19.1.27), the amplification factor (for constant v) is

$$\xi = 1 - \left|\frac{v\Delta t}{\Delta x}\right| (1-\cos k\Delta x) - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.28}$$

$$|\xi|^2 = 1 - 2\left|\frac{v\Delta t}{\Delta x}\right|\left(1-\left|\frac{v\Delta t}{\Delta x}\right|\right)(1-\cos k\Delta x) \tag{19.1.29}$$

So the stability criterion $|\xi|^2 \le 1$ is (again) simply the Courant condition (19.1.17).

There are various ways of improving the accuracy of first-order upwind differencing. In the continuum equation, material originally a distance $v\Delta t$ away

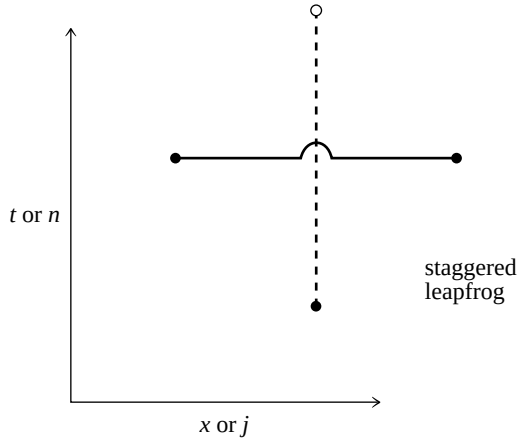


Figure 19.1.5. Representation of the staggered leapfrog differencing scheme. Note that information from two previous time slices is used in obtaining the desired point. This scheme is second-order accurate in both space and time.

arrives at a given point after a time interval Δt. In the first-order method, the material always arrives from Δx away. If $v\Delta t \ll \Delta x$ (to insure accuracy), this can cause a large error. One way of reducing this error is to interpolate u between $j-1$ and j before transporting it. This gives effectively a second-order method. Various schemes for second-order upwind differencing are discussed and compared in [2-3].

Second-Order Accuracy in Time

When using a method that is first-order accurate in time but second-order accurate in space, one generally has to take $v\Delta t$ significantly smaller than Δx to achieve desired accuracy, say, by at least a factor of 5. Thus the Courant condition is not actually the limiting factor with such schemes in practice. However, there are schemes that are second-order accurate in both space and time, and these can often be pushed right to their stability limit, with correspondingly smaller computation times.

For example, the *staggered leapfrog* method for the conservation equation (19.1.1) is defined as follows (Figure 19.1.5): Using the values of u^n at time t^n, compute the fluxes F_j^n. Then compute new values u^{n+1} using the time-centered values of the fluxes:

$$u_j^{n+1} - u_j^{n-1} = -\frac{\Delta t}{\Delta x}(F_{j+1}^n - F_{j-1}^n) \tag{19.1.30}$$

The name comes from the fact that the time levels in the time derivative term "leapfrog" over the time levels in the space derivative term. The method requires that u^{n-1} and u^n be stored to compute u^{n+1}.

For our simple model equation (19.1.6), staggered leapfrog takes the form

$$u_j^{n+1} - u_j^{n-1} = -\frac{v\Delta t}{\Delta x}(u_{j+1}^n - u_{j-1}^n) \tag{19.1.31}$$

The von Neumann stability analysis now gives a quadratic equation for ξ, rather than a linear one, because of the occurrence of three consecutive powers of ξ when the

form (19.1.12) for an eigenmode is substituted into equation (19.1.31),

$$\xi^2 - 1 = -2i\xi \frac{v\Delta t}{\Delta x} \sin k\Delta x \tag{19.1.32}$$

whose solution is

$$\xi = -i\frac{v\Delta t}{\Delta x}\sin k\Delta x \pm \sqrt{1 - \left(\frac{v\Delta t}{\Delta x}\sin k\Delta x\right)^2} \tag{19.1.33}$$

Thus the Courant condition is again required for stability. In fact, in equation (19.1.33), $|\xi|^2 = 1$ for any $v\Delta t \le \Delta x$. This is the great advantage of the staggered leapfrog method: There is no amplitude dissipation.

Staggered leapfrog differencing of equations like (19.1.20) is most transparent if the variables are centered on appropriate half-mesh points:

$$\begin{aligned} r^n_{j+1/2} &\equiv v\left.\frac{\partial u}{\partial x}\right|^n_{j+1/2} = v\frac{u^n_{j+1} - u^n_j}{\Delta x} \\ s^{n+1/2}_j &\equiv \left.\frac{\partial u}{\partial t}\right|^{n+1/2}_j = \frac{u^{n+1}_j - u^n_j}{\Delta t} \end{aligned} \tag{19.1.34}$$

This is purely a notational convenience: we can think of the mesh on which r and s are defined as being twice as fine as the mesh on which the original variable u is defined. The leapfrog differencing of equation (19.1.20) is

$$\begin{aligned} \frac{r^{n+1}_{j+1/2} - r^n_{j+1/2}}{\Delta t} &= \frac{s^{n+1/2}_{j+1} - s^{n+1/2}_j}{\Delta x} \\ \frac{s^{n+1/2}_j - s^{n-1/2}_j}{\Delta t} &= v\frac{r^n_{j+1/2} - r^n_{j-1/2}}{\Delta x} \end{aligned} \tag{19.1.35}$$

If you substitute equation (19.1.22) in equation (19.1.35), you will find that once again the Courant condition is required for stability, and that there is no amplitude dissipation when it is satisfied.

If we substitute equation (19.1.34) in equation (19.1.35), we find that equation (19.1.35) is equivalent to

$$\frac{u^{n+1}_j - 2u^n_j + u^{n-1}_j}{(\Delta t)^2} = v^2\frac{u^n_{j+1} - 2u^n_j + u^n_{j-1}}{(\Delta x)^2} \tag{19.1.36}$$

This is just the "usual" second-order differencing of the wave equation (19.1.2). We see that it is a two-level scheme, requiring both u^n and u^{n-1} to obtain u^{n+1}. In equation (19.1.35) this shows up as both $s^{n-1/2}$ and r^n being needed to advance the solution.

For equations more complicated than our simple model equation, especially nonlinear equations, the leapfrog method usually becomes unstable when the gradients get large. The instability is related to the fact that odd and even mesh points are completely decoupled, like the black and white squares of a chess board, as shown

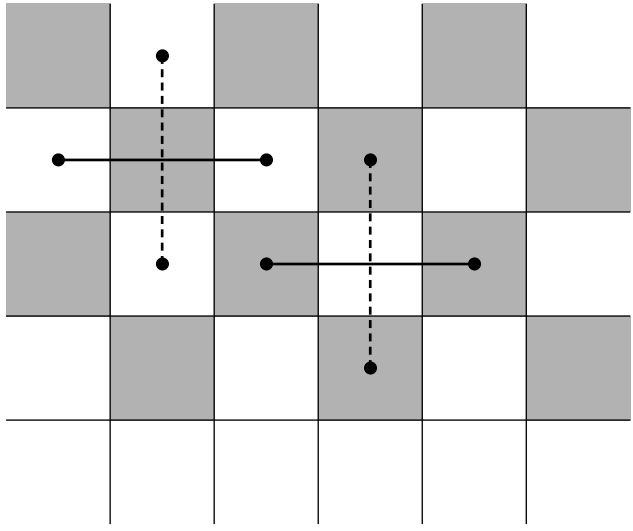


Figure 19.1.6. Origin of mesh-drift instabilities in a staggered leapfrog scheme. If the mesh points are imagined to lie in the squares of a chess board, then white squares couple to themselves, black to themselves, but there is no coupling between white and black. The fix is to introduce a small diffusive mesh-coupling piece.

in Figure 19.1.6. This mesh drifting instability is cured by coupling the two meshes through a numerical viscosity term, e.g., adding to the right side of (19.1.31) a small coefficient ($\ll 1$) times $u^n_{j+1} - 2u^n_j + u^n_{j-1}$. For more on stabilizing difference schemes by adding numerical dissipation, see, e.g., [4].

The *Two-Step Lax-Wendroff* scheme is a second-order in time method that avoids large numerical dissipation and mesh drifting. One defines intermediate values $u_{j+1/2}$ at the half timesteps $t_{n+1/2}$ and the half mesh points $x_{j+1/2}$. These are calculated by the Lax scheme:

$$u^{n+1/2}_{j+1/2} = \frac{1}{2}(u^n_{j+1} + u^n_j) - \frac{\Delta t}{2\Delta x}(F^n_{j+1} - F^n_j) \tag{19.1.37}$$

Using these variables, one calculates the fluxes $F^{n+1/2}_{j+1/2}$. Then the updated values u^{n+1}_j are calculated by the properly centered expression

$$u^{n+1}_j = u^n_j - \frac{\Delta t}{\Delta x}\left(F^{n+1/2}_{j+1/2} - F^{n+1/2}_{j-1/2}\right) \tag{19.1.38}$$

The provisional values $u^{n+1/2}_{j+1/2}$ are now discarded. (See Figure 19.1.7.)

Let us investigate the stability of this method for our model advective equation, where $F = vu$. Substitute (19.1.37) in (19.1.38) to get

$$\begin{aligned} u^{n+1}_j = u^n_j - \alpha \Bigg[& \frac{1}{2}(u^n_{j+1} + u^n_j) - \frac{1}{2}\alpha(u^n_{j+1} - u^n_j) \\ & - \frac{1}{2}(u^n_j + u^n_{j-1}) + \frac{1}{2}\alpha(u^n_j - u^n_{j-1}) \Bigg] \end{aligned} \tag{19.1.39}$$

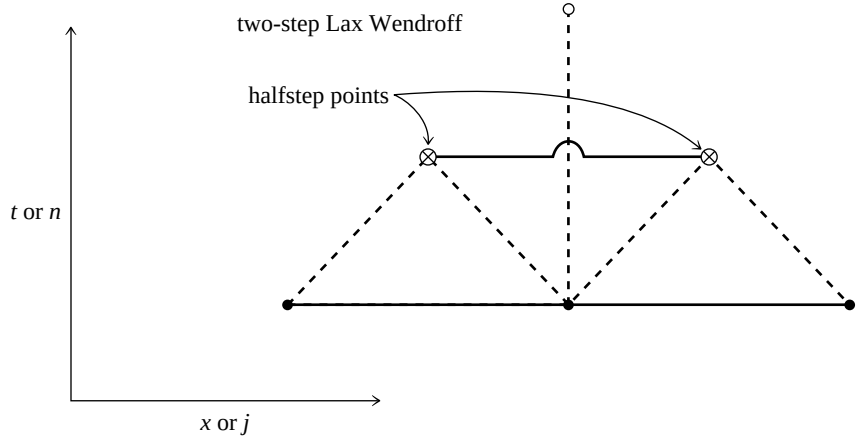


Figure 19.1.7. Representation of the two-step Lax-Wendroff differencing scheme. Two halfstep points ($\otimes$) are calculated by the Lax method. These, plus one of the original points, produce the new point via staggered leapfrog. Halfstep points are used only temporarily and do not require storage allocation on the grid. This scheme is second-order accurate in both space and time.

where

$$\alpha \equiv \frac{v\Delta t}{\Delta x} \tag{19.1.40}$$

Then

$$\xi = 1 - i\alpha \sin k\Delta x - \alpha^2(1 - \cos k\Delta x) \tag{19.1.41}$$

so

$$|\xi|^2 = 1 - \alpha^2(1 - \alpha^2)(1 - \cos k\Delta x)^2 \tag{19.1.42}$$

The stability criterion $|\xi|^2 \leq 1$ is therefore $\alpha^2 \leq 1$, or $v\Delta t \leq \Delta x$ as usual. Incidentally, you should not think that the Courant condition is the only stability requirement that ever turns up in PDEs. It keeps doing so in our model examples just because those examples are so simple in form. The method of analysis is, however, general.

Except when $\alpha = 1$, $|\xi|^2 < 1$ in (19.1.42), so some amplitude damping does occur. The effect is relatively small, however, for wavelengths large compared with the mesh size Δx. If we expand (19.1.42) for small $k\Delta x$, we find

$$|\xi|^2 = 1 - \alpha^2(1 - \alpha^2)\frac{(k\Delta x)^4}{4} + \dots \tag{19.1.43}$$

The departure from unity occurs only at fourth order in k. This should be contrasted with equation (19.1.16) for the Lax method, which shows that

$$|\xi|^2 = 1 - (1 - \alpha^2)(k\Delta x)^2 + \dots \tag{19.1.44}$$

for small $k\Delta x$.

In summary, our recommendation for initial value problems that can be cast in flux-conservative form, and especially problems related to the wave equation, is to use the staggered leapfrog method when possible. We have personally had better success with it than with the Two-Step Lax-Wendroff method. For problems sensitive to transport errors, upwind differencing or one of its refinements should be considered.

Fluid Dynamics with Shocks

As we alluded to earlier, the treatment of fluid dynamics problems with shocks has become a very complicated and very sophisticated subject. All we can attempt to do here is to guide you to some starting points in the literature.

There are basically three important general methods for handling shocks. The oldest and simplest method, invented by von Neumann and Richtmyer, is to add *artificial viscosity* to the equations, modeling the way Nature uses real viscosity to smooth discontinuities. A good starting point for trying out this method is the differencing scheme in §12.11 of [1]. This scheme is excellent for nearly all problems in one spatial dimension.

The second method combines a high-order differencing scheme that is accurate for smooth flows with a low order scheme that is very dissipative and can smooth the shocks. Typically, various upwind differencing schemes are combined using weights chosen to zero the low order scheme unless steep gradients are present, and also chosen to enforce various "monotonicity" constraints that prevent nonphysical oscillations from appearing in the numerical solution. References [2-3,5] are a good place to start with these methods.

The third, and potentially most powerful method, is Godunov's approach. Here one gives up the simple linearization inherent in finite differencing based on Taylor series and includes the nonlinearity of the equations explicitly. There is an analytic solution for the evolution of two uniform states of a fluid separated by a discontinuity, the Riemann shock problem. Godunov's idea was to approximate the fluid by a large number of cells of uniform states, and piece them together using the Riemann solution. There have been many generalizations of Godunov's approach, of which the most powerful is probably the PPM method [6].

Readable reviews of all these methods, discussing the difficulties arising when one-dimensional methods are generalized to multidimensions, are given in [7-9].

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 4.

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [1]

Centrella, J., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 54, pp. 229–249, Appendix B. [2]

Hawley, J.F., Smarr, L.L., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 55, pp. 211–246, §2c. [3]

Kreiss, H.-O. 1978, *Numerical Methods for Solving Time-Dependent Problems for Partial Differential Equations* (Montreal: University of Montreal Press), pp. 66ff. [4]

Harten, A., Lax, P.D., and Van Leer, B. 1983, *SIAM Review*, vol. 25, pp. 36–61. [5]

Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 174–201. [6]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [7]
Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 115–173. [8]
Rizzi, A., and Engquist, B. 1987, *Journal of Computational Physics*, vol. 72, pp. 1–69. [9]

19.2 Diffusive Initial Value Problems

Recall the model parabolic equation, the diffusion equation in one space dimension,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.2.1}$$

where D is the diffusion coefficient. Actually, this equation is a flux-conservative equation of the form considered in the previous section, with

$$F = -D\frac{\partial u}{\partial x} \tag{19.2.2}$$

the flux in the x-direction. We will assume $D \geq 0$, otherwise equation (19.2.1) has physically unstable solutions: A small disturbance evolves to become more and more concentrated instead of dispersing. (Don't make the mistake of trying to find a stable differencing scheme for a problem whose underlying PDEs are themselves unstable!)

Even though (19.2.1) is of the form already considered, it is useful to consider it as a model in its own right. The particular form of flux (19.2.2), and its direct generalizations, occur quite frequently in practice. Moreover, we have already seen that numerical viscosity and artificial viscosity can introduce diffusive pieces like the right-hand side of (19.2.1) in many other situations.

Consider first the case when D is a constant. Then the equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \tag{19.2.3}$$

can be differenced in the obvious way:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D\left[\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right] \tag{19.2.4}$$

This is the FTCS scheme again, except that it is a second derivative that has been differenced on the right-hand side. But this makes a world of difference! The FTCS scheme was unstable for the hyperbolic equation; however, a quick calculation shows that the amplification factor for equation (19.2.4) is

$$\xi = 1 - \frac{4D\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\Delta x}{2}\right) \tag{19.2.5}$$

The requirement $|\xi| \leq 1$ leads to the stability criterion

$$\frac{2D\Delta t}{(\Delta x)^2} \leq 1 \tag{19.2.6}$$

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [7]
Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 115–173. [8]
Rizzi, A., and Engquist, B. 1987, *Journal of Computational Physics*, vol. 72, pp. 1–69. [9]

19.2 Diffusive Initial Value Problems

Recall the model parabolic equation, the diffusion equation in one space dimension,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.2.1}$$

where D is the diffusion coefficient. Actually, this equation is a flux-conservative equation of the form considered in the previous section, with

$$F = -D\frac{\partial u}{\partial x} \tag{19.2.2}$$

the flux in the x-direction. We will assume $D \geq 0$, otherwise equation (19.2.1) has physically unstable solutions: A small disturbance evolves to become more and more concentrated instead of dispersing. (Don't make the mistake of trying to find a stable differencing scheme for a problem whose underlying PDEs are themselves unstable!)

Even though (19.2.1) is of the form already considered, it is useful to consider it as a model in its own right. The particular form of flux (19.2.2), and its direct generalizations, occur quite frequently in practice. Moreover, we have already seen that numerical viscosity and artificial viscosity can introduce diffusive pieces like the right-hand side of (19.2.1) in many other situations.

Consider first the case when D is a constant. Then the equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \tag{19.2.3}$$

can be differenced in the obvious way:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D\left[\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right] \tag{19.2.4}$$

This is the FTCS scheme again, except that it is a second derivative that has been differenced on the right-hand side. But this makes a world of difference! The FTCS scheme was unstable for the hyperbolic equation; however, a quick calculation shows that the amplification factor for equation (19.2.4) is

$$\xi = 1 - \frac{4D\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\Delta x}{2}\right) \tag{19.2.5}$$

The requirement $|\xi| \leq 1$ leads to the stability criterion

$$\frac{2D\Delta t}{(\Delta x)^2} \leq 1 \tag{19.2.6}$$

The physical interpretation of the restriction (19.2.6) is that the maximum allowed timestep is, up to a numerical factor, the diffusion time across a cell of width Δx.

More generally, the diffusion time τ across a spatial scale of size λ is of order

$$\tau \sim \frac{\lambda^2}{D} \tag{19.2.7}$$

Usually we are interested in modeling accurately the evolution of features with spatial scales $\lambda \gg \Delta x$. If we are limited to timesteps satisfying (19.2.6), we will need to evolve through of order $\lambda^2/(\Delta x)^2$ steps before things start to happen on the scale of interest. This number of steps is usually prohibitive. We must therefore find a stable way of taking timesteps comparable to, or perhaps — for accuracy — somewhat smaller than, the time scale of (19.2.7).

This goal poses an immediate "philosophical" question. Obviously the large timesteps that we propose to take are going to be woefully inaccurate for the small scales that we have decided not to be interested in. We want those scales to do something stable, "innocuous," and perhaps not too physically unreasonable. We want to build this innocuous behavior into our differencing scheme. What should it be?

There are two different answers, each of which has its pros and cons. The first answer is to seek a differencing scheme that drives small-scale features to their *equilibrium* forms, e.g., satisfying equation (19.2.3) with the left-hand side set to zero. This answer generally makes the best physical sense; but, as we will see, it leads to a differencing scheme ("fully implicit") that is only *first-order* accurate in time for the scales that we are interested in. The second answer is to let small-scale features *maintain* their initial amplitudes, so that the evolution of the larger-scale features of interest takes place superposed with a kind of "frozen in" (though fluctuating) background of small-scale stuff. This answer gives a differencing scheme ("Crank-Nicholson") that is *second-order* accurate in time. Toward the end of an evolution calculation, however, one might want to switch over to some steps of the other kind, to drive the small-scale stuff into equilibrium. Let us now see where these distinct differencing schemes come from:

Consider the following differencing of (19.2.3),

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D\left[\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2}\right] \tag{19.2.8}$$

This is exactly like the FTCS scheme (19.2.4), except that the spatial derivatives on the right-hand side are evaluated at timestep $n+1$. Schemes with this character are called *fully implicit* or *backward time*, by contrast with FTCS (which is called *fully explicit*). To solve equation (19.2.8) one has to solve a set of simultaneous linear equations at each timestep for the u_j^{n+1}. Fortunately, this is a simple problem because the system is tridiagonal: Just group the terms in equation (19.2.8) appropriately:

$$-\alpha u_{j-1}^{n+1} + (1+2\alpha)u_j^{n+1} - \alpha u_{j+1}^{n+1} = u_j^n, \qquad j = 1, 2 \ldots J-1 \tag{19.2.9}$$

where

$$\alpha \equiv \frac{D\Delta t}{(\Delta x)^2} \tag{19.2.10}$$

Supplemented by Dirichlet or Neumann boundary conditions at $j = 0$ and $j = J$, equation (19.2.9) is clearly a tridiagonal system, which can easily be solved at each timestep by the method of §2.4.

What is the behavior of (19.2.8) for very large timesteps? The answer is seen most clearly in (19.2.9), in the limit $\alpha \to \infty$ ($\Delta t \to \infty$). Dividing by α, we see that the difference equations are just the finite-difference form of the equilibrium equation

$$\frac{\partial^2 u}{\partial x^2} = 0 \tag{19.2.11}$$

What about stability? The amplification factor for equation (19.2.8) is

$$\xi = \frac{1}{1 + 4\alpha \sin^2\left(\dfrac{k\Delta x}{2}\right)} \tag{19.2.12}$$

Clearly $|\xi| < 1$ for any stepsize Δt. The scheme is unconditionally stable. The details of the small-scale evolution from the initial conditions are obviously inaccurate for large Δt. But, as advertised, the correct equilibrium solution is obtained. This is the characteristic feature of implicit methods.

Here, on the other hand, is how one gets to the second of our above philosophical answers, combining the stability of an implicit method with the accuracy of a method that is second-order in both space and time. Simply form the average of the explicit and implicit FTCS schemes:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{D}{2}\left[\frac{(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (u_{j+1}^n - 2u_j^n + u_{j-1}^n)}{(\Delta x)^2}\right] \tag{19.2.13}$$

Here both the left- and right-hand sides are centered at timestep $n + \frac{1}{2}$, so the method is second-order accurate in time as claimed. The amplification factor is

$$\xi = \frac{1 - 2\alpha \sin^2\left(\dfrac{k\Delta x}{2}\right)}{1 + 2\alpha \sin^2\left(\dfrac{k\Delta x}{2}\right)} \tag{19.2.14}$$

so the method is stable for any size Δt. This scheme is called the *Crank-Nicholson* scheme, and is our recommended method for any simple diffusion problem (perhaps supplemented by a few fully implicit steps at the end). (See Figure 19.2.1.)

Now turn to some generalizations of the simple diffusion equation (19.2.3). Suppose first that the diffusion coefficient D is not constant, say $D = D(x)$. We can adopt either of two strategies. First, we can make an analytic change of variable

$$y = \int \frac{dx}{D(x)} \tag{19.2.15}$$

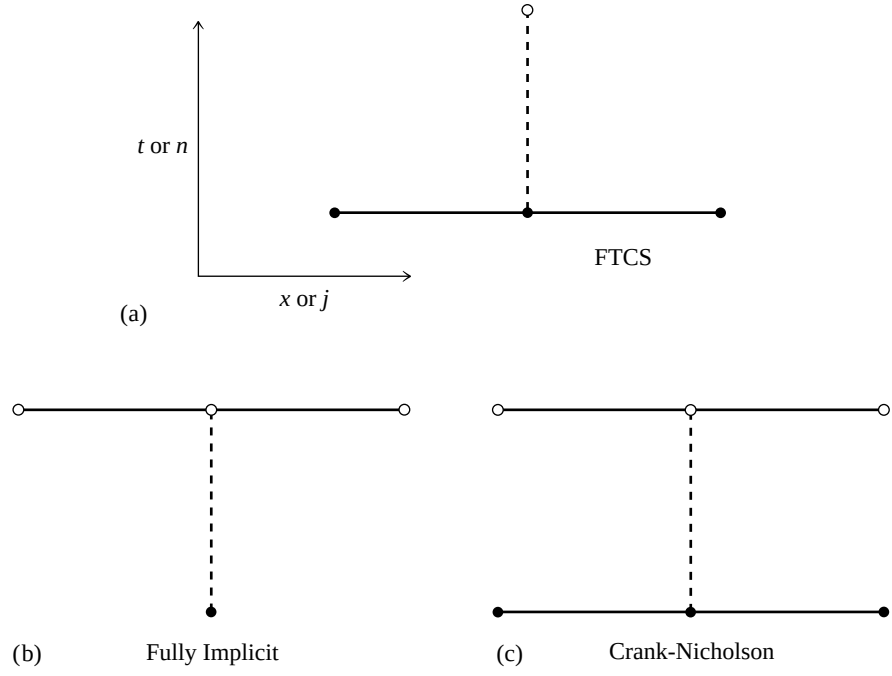


Figure 19.2.1. Three differencing schemes for diffusive problems (shown as in Figure 19.1.2). (a) Forward Time Center Space is first-order accurate, but stable only for sufficiently small timesteps. (b) Fully Implicit is stable for arbitrarily large timesteps, but is still only first-order accurate. (c) Crank-Nicholson is second-order accurate, and is usually stable for large timesteps.

Then

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x} D(x) \frac{\partial u}{\partial x} \tag{19.2.16}$$

becomes

$$\frac{\partial u}{\partial t} = \frac{1}{D(y)} \frac{\partial^2 u}{\partial y^2} \tag{19.2.17}$$

and we evaluate D at the appropriate y_j. Heuristically, the stability criterion (19.2.6) in an explicit scheme becomes

$$\Delta t \leq \min_j \left[\frac{(\Delta y)^2}{2D_j^{-1}} \right] \tag{19.2.18}$$

Note that constant spacing Δy in y does not imply constant spacing in x.

An alternative method that does not require analytically tractable forms for D is simply to difference equation (19.2.16) as it stands, centering everything appropriately. Thus the FTCS method becomes

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{D_{j+1/2}(u_{j+1}^n - u_j^n) - D_{j-1/2}(u_j^n - u_{j-1}^n)}{(\Delta x)^2} \tag{19.2.19}$$

where

$$D_{j+1/2} \equiv D(x_{j+1/2}) \tag{19.2.20}$$

and the heuristic stability criterion is

$$\Delta t \le \min_j \left[\frac{(\Delta x)^2}{2D_{j+1/2}} \right] \tag{19.2.21}$$

The Crank-Nicholson method can be generalized similarly.

The second complication one can consider is a nonlinear diffusion problem, for example where $D = D(u)$. Explicit schemes can be generalized in the obvious way. For example, in equation (19.2.19) write

$$D_{j+1/2} = \frac{1}{2}\left[D(u^n_{j+1}) + D(u^n_j) \right] \tag{19.2.22}$$

Implicit schemes are not as easy. The replacement (19.2.22) with $n \to n+1$ leaves us with a nasty set of coupled nonlinear equations to solve at each timestep. Often there is an easier way: If the form of $D(u)$ allows us to integrate

$$dz = D(u)du \tag{19.2.23}$$

analytically for $z(u)$, then the right-hand side of (19.2.1) becomes $\partial^2 z/\partial x^2$, which we difference implicitly as

$$\frac{z^{n+1}_{j+1} - 2z^{n+1}_j + z^{n+1}_{j-1}}{(\Delta x)^2} \tag{19.2.24}$$

Now linearize each term on the right-hand side of equation (19.2.24), for example

$$\begin{aligned} z^{n+1}_j \equiv z(u^{n+1}_j) &= z(u^n_j) + (u^{n+1}_j - u^n_j)\left.\frac{\partial z}{\partial u}\right|_{j,n} \\ &= z(u^n_j) + (u^{n+1}_j - u^n_j) D(u^n_j) \end{aligned} \tag{19.2.25}$$

This reduces the problem to tridiagonal form again and in practice usually retains the stability advantages of fully implicit differencing.

Schrödinger Equation

Sometimes the physical problem being solved imposes constraints on the differencing scheme that we have not yet taken into account. For example, consider the time-dependent Schrödinger equation of quantum mechanics. This is basically a parabolic equation for the evolution of a complex quantity ψ. For the scattering of a wavepacket by a one-dimensional potential $V(x)$, the equation has the form

$$i\frac{\partial \psi}{\partial t} = -\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi \tag{19.2.26}$$

(Here we have chosen units so that Planck's constant $\hbar = 1$ and the particle mass $m = 1/2$.) One is given the initial wavepacket, $\psi(x, t = 0)$, together with boundary

conditions that $\psi \to 0$ at $x \to \pm\infty$. Suppose we content ourselves with first-order accuracy in time, but want to use an implicit scheme, for stability. A slight generalization of (19.2.8) leads to

$$i\left[\frac{\psi_j^{n+1}-\psi_j^n}{\Delta t}\right] = -\left[\frac{\psi_{j+1}^{n+1}-2\psi_j^{n+1}+\psi_{j-1}^{n+1}}{(\Delta x)^2}\right] + V_j\psi_j^{n+1} \tag{19.2.27}$$

for which

$$\xi = \frac{1}{1+i\left[\dfrac{4\Delta t}{(\Delta x)^2}\sin^2\left(\dfrac{k\Delta x}{2}\right)+V_j\Delta t\right]} \tag{19.2.28}$$

This is unconditionally stable, but unfortunately is not *unitary*. The underlying physical problem requires that the total probability of finding the particle somewhere remains unity. This is represented formally by the modulus-square norm of ψ remaining unity:

$$\int_{-\infty}^{\infty}|\psi|^2 dx = 1 \tag{19.2.29}$$

The initial wave function $\psi(x,0)$ is normalized to satisfy (19.2.29). The Schrödinger equation (19.2.26) then guarantees that this condition is satisfied at all later times.

Let us write equation (19.2.26) in the form

$$i\frac{\partial\psi}{\partial t} = H\psi \tag{19.2.30}$$

where the operator H is

$$H = -\frac{\partial^2}{\partial x^2} + V(x) \tag{19.2.31}$$

The formal solution of equation (19.2.30) is

$$\psi(x,t) = e^{-iHt}\psi(x,0) \tag{19.2.32}$$

where the exponential of the operator is defined by its power series expansion.

The unstable explicit FTCS scheme approximates (19.2.32) as

$$\psi_j^{n+1} = (1-iH\Delta t)\psi_j^n \tag{19.2.33}$$

where H is represented by a centered finite-difference approximation in x. The stable implicit scheme (19.2.27) is, by contrast,

$$\psi_j^{n+1} = (1+iH\Delta t)^{-1}\psi_j^n \tag{19.2.34}$$

These are both first-order accurate in time, as can be seen by expanding equation (19.2.32). However, neither operator in (19.2.33) or (19.2.34) is unitary.

The correct way to difference Schrödinger's equation [1,2] is to use *Cayley's form* for the finite-difference representation of e^{-iHt}, which is second-order accurate *and* unitary:

$$e^{-iHt} \simeq \frac{1 - \frac{1}{2}iH\Delta t}{1 + \frac{1}{2}iH\Delta t} \tag{19.2.35}$$

In other words,

$$\left(1 + \tfrac{1}{2}iH\Delta t\right)\psi_j^{n+1} = \left(1 - \tfrac{1}{2}iH\Delta t\right)\psi_j^n \tag{19.2.36}$$

On replacing H by its finite-difference approximation in x, we have a complex tridiagonal system to solve. The method is stable, unitary, and second-order accurate in space and time. In fact, it is simply the Crank-Nicholson method once again!

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 2.

Goldberg, A., Schey, H.M., and Schwartz, J.L. 1967, *American Journal of Physics*, vol. 35, pp. 177–186. [1]

Galbraith, I., Ching, Y.S., and Abraham, E. 1984, *American Journal of Physics*, vol. 52, pp. 60–68. [2]

19.3 Initial Value Problems in Multidimensions

The methods described in §19.1 and §19.2 for problems in $1 + 1$ dimension (one space and one time dimension) can easily be generalized to $N + 1$ dimensions. However, the computing power necessary to solve the resulting equations is enormous. If you have solved a one-dimensional problem with 100 spatial grid points, solving the two-dimensional version with 100×100 mesh points requires *at least* 100 times as much computing. You generally have to be content with very modest spatial resolution in multidimensional problems.

Indulge us in offering a bit of advice about the development and testing of multidimensional PDE codes: You should always first run your programs on *very small* grids, e.g., 8×8, even though the resulting accuracy is so poor as to be useless. When your program is all debugged and demonstrably stable, *then* you can increase the grid size to a reasonable one and start looking at the results. We have actually heard someone protest, "my program would be unstable for a crude grid, but I am sure the instability will go away on a larger grid." That is nonsense of a most pernicious sort, evidencing total confusion between accuracy and stability. In fact, new instabilities sometimes do show up on *larger* grids; but old instabilities never (in our experience) just go away.

Forced to live with modest grid sizes, some people recommend going to higher-order methods in an attempt to improve accuracy. This is very dangerous. Unless the solution you are looking for is known to be smooth, and the high-order method you

The correct way to difference Schrödinger's equation [1,2] is to use *Cayley's form* for the finite-difference representation of e^{-iHt}, which is second-order accurate *and* unitary:

$$e^{-iHt} \simeq \frac{1 - \frac{1}{2}iH\Delta t}{1 + \frac{1}{2}iH\Delta t} \tag{19.2.35}$$

In other words,

$$\left(1 + \tfrac{1}{2}iH\Delta t\right)\psi_j^{n+1} = \left(1 - \tfrac{1}{2}iH\Delta t\right)\psi_j^n \tag{19.2.36}$$

On replacing H by its finite-difference approximation in x, we have a complex tridiagonal system to solve. The method is stable, unitary, and second-order accurate in space and time. In fact, it is simply the Crank-Nicholson method once again!

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 2.

Goldberg, A., Schey, H.M., and Schwartz, J.L. 1967, *American Journal of Physics*, vol. 35, pp. 177–186. [1]

Galbraith, I., Ching, Y.S., and Abraham, E. 1984, *American Journal of Physics*, vol. 52, pp. 60–68. [2]

19.3 Initial Value Problems in Multidimensions

The methods described in §19.1 and §19.2 for problems in $1 + 1$ dimension (one space and one time dimension) can easily be generalized to $N + 1$ dimensions. However, the computing power necessary to solve the resulting equations is enormous. If you have solved a one-dimensional problem with 100 spatial grid points, solving the two-dimensional version with 100×100 mesh points requires *at least* 100 times as much computing. You generally have to be content with very modest spatial resolution in multidimensional problems.

Indulge us in offering a bit of advice about the development and testing of multidimensional PDE codes: You should always first run your programs on *very small* grids, e.g., 8×8, even though the resulting accuracy is so poor as to be useless. When your program is all debugged and demonstrably stable, *then* you can increase the grid size to a reasonable one and start looking at the results. We have actually heard someone protest, "my program would be unstable for a crude grid, but I am sure the instability will go away on a larger grid." That is nonsense of a most pernicious sort, evidencing total confusion between accuracy and stability. In fact, new instabilities sometimes do show up on *larger* grids; but old instabilities never (in our experience) just go away.

Forced to live with modest grid sizes, some people recommend going to higher-order methods in an attempt to improve accuracy. This is very dangerous. Unless the solution you are looking for is known to be smooth, and the high-order method you

are using is known to be extremely stable, we do not recommend anything higher than second-order in time (for sets of first-order equations). For spatial differencing, we recommend the order of the underlying PDEs, perhaps allowing second-order spatial differencing for first-order-in-space PDEs. When you increase the order of a differencing method to greater than the order of the original PDEs, you introduce spurious solutions to the difference equations. This does not create a problem if they all happen to decay exponentially; otherwise you are going to see all hell break loose!

Lax Method for a Flux-Conservative Equation

As an example, we show how to generalize the Lax method (19.1.15) to two dimensions for the conservation equation

$$\frac{\partial u}{\partial t} = -\nabla \cdot \mathbf{F} = -\left(\frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y}\right) \tag{19.3.1}$$

Use a spatial grid with

$$\begin{aligned} x_j &= x_0 + j\Delta \\ y_l &= y_0 + l\Delta \end{aligned} \tag{19.3.2}$$

We have chosen $\Delta x = \Delta y \equiv \Delta$ for simplicity. Then the Lax scheme is

$$\begin{aligned} u_{j,l}^{n+1} = \frac{1}{4}(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n) \\ - \frac{\Delta t}{2\Delta}(F_{j+1,l}^n - F_{j-1,l}^n + F_{j,l+1}^n - F_{j,l-1}^n) \end{aligned} \tag{19.3.3}$$

Note that as an abbreviated notation F_{j+1} and F_{j-1} refer to F_x, while F_{l+1} and F_{l-1} refer to F_y.

Let us carry out a stability analysis for the model advective equation (analog of 19.1.6) with

$$F_x = v_x u, \qquad F_y = v_y u \tag{19.3.4}$$

This requires an eigenmode with two dimensions in space, though still only a simple dependence on powers of ξ in time,

$$u_{j,l}^n = \xi^n e^{ik_x j\Delta} e^{ik_y l\Delta} \tag{19.3.5}$$

Substituting in equation (19.3.3), we find

$$\xi = \frac{1}{2}(\cos k_x\Delta + \cos k_y\Delta) - i\alpha_x \sin k_x\Delta - i\alpha_y \sin k_y\Delta \tag{19.3.6}$$

where

$$\alpha_x = \frac{v_x\Delta t}{\Delta}, \qquad \alpha_y = \frac{v_y\Delta t}{\Delta} \tag{19.3.7}$$

The expression for $|\xi|^2$ can be manipulated into the form

$$|\xi|^2 = 1 - (\sin^2 k_x\Delta + \sin^2 k_y\Delta)\left[\frac{1}{2} - (\alpha_x^2 + \alpha_y^2)\right] - \frac{1}{4}(\cos k_x\Delta - \cos k_y\Delta)^2 - (\alpha_y \sin k_x\Delta - \alpha_x \sin k_y\Delta)^2 \tag{19.3.8}$$

The last two terms are negative, and so the stability requirement $|\xi|^2 \le 1$ becomes

$$\frac{1}{2} - (\alpha_x^2 + \alpha_y^2) \ge 0 \tag{19.3.9}$$

or

$$\Delta t \le \frac{\Delta}{\sqrt{2}(v_x^2 + v_y^2)^{1/2}} \tag{19.3.10}$$

This is an example of the general result for the N-dimensional Courant condition: If $|v|$ is the maximum propagation velocity in the problem, then

$$\Delta t \le \frac{\Delta}{\sqrt{N}|v|} \tag{19.3.11}$$

is the Courant condition.

Diffusion Equation in Multidimensions

Let us consider the two-dimensional diffusion equation,

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{19.3.12}$$

An explicit method, such as FTCS, can be generalized from the one-dimensional case in the obvious way. However, we have seen that diffusive problems are usually best treated implicitly. Suppose we try to implement the Crank-Nicholson scheme in two dimensions. This would give us

$$u_{j,l}^{n+1} = u_{j,l}^n + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1} + \delta_x^2 u_{j,l}^n + \delta_y^2 u_{j,l}^{n+1} + \delta_y^2 u_{j,l}^n\right) \tag{19.3.13}$$

Here

$$\alpha \equiv \frac{D\Delta t}{\Delta^2} \qquad \Delta \equiv \Delta x = \Delta y \tag{19.3.14}$$

$$\delta_x^2 u_{j,l}^n \equiv u_{j+1,l}^n - 2u_{j,l}^n + u_{j-1,l}^n \tag{19.3.15}$$

and similarly for $\delta_y^2 u_{j,l}^n$. This is certainly a viable scheme; the problem arises in solving the coupled linear equations. Whereas in one space dimension the system was tridiagonal, that is no longer true, though the matrix is still very sparse. One possibility is to use a suitable sparse matrix technique (see §2.7 and §19.0).

Another possibility, which we generally prefer, is a slightly different way of generalizing the Crank-Nicholson algorithm. It is still second-order accurate in time and space, and unconditionally stable, but the equations are easier to solve than

(19.3.13). Called the *alternating-direction implicit method (ADI)*, this embodies the powerful concept of *operator splitting* or *time splitting*, about which we will say more below. Here, the idea is to divide each timestep into two steps of size $\Delta t/2$. In each substep, a different dimension is treated implicitly:

$$\begin{aligned} u_{j,l}^{n+1/2} &= u_{j,l}^{n} + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n}\right) \\ u_{j,l}^{n+1} &= u_{j,l}^{n+1/2} + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n+1}\right) \end{aligned} \tag{19.3.16}$$

The advantage of this method is that each substep requires only the solution of a simple tridiagonal system.

Operator Splitting Methods Generally

The basic idea of operator splitting, which is also called *time splitting* or *the method of fractional steps*, is this: Suppose you have an initial value equation of the form

$$\frac{\partial u}{\partial t} = \mathcal{L}u \tag{19.3.17}$$

where $\mathcal{L}$ is some operator. While $\mathcal{L}$ is not necessarily linear, suppose that it can at least be written as a linear sum of m pieces, which act additively on u,

$$\mathcal{L}u = \mathcal{L}_1 u + \mathcal{L}_2 u + \cdots + \mathcal{L}_m u \tag{19.3.18}$$

Finally, suppose that for *each* of the pieces, you already know a differencing scheme for updating the variable u from timestep n to timestep $n+1$, valid if that piece of the operator were the *only* one on the right-hand side. We will write these updatings symbolically as

$$\begin{aligned} u^{n+1} &= \mathcal{U}_1(u^n, \Delta t) \\ u^{n+1} &= \mathcal{U}_2(u^n, \Delta t) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^n, \Delta t) \end{aligned} \tag{19.3.19}$$

Now, one form of operator splitting would be to get from n to $n+1$ by the following sequence of updatings:

$$\begin{aligned} u^{n+(1/m)} &= \mathcal{U}_1(u^n, \Delta t) \\ u^{n+(2/m)} &= \mathcal{U}_2(u^{n+(1/m)}, \Delta t) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t) \end{aligned} \tag{19.3.20}$$

For example, a combined advective-diffusion equation, such as

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + D\frac{\partial^2 u}{\partial x^2} \tag{19.3.21}$$

might profitably use an explicit scheme for the advective term combined with a Crank-Nicholson or other implicit scheme for the diffusion term.

The alternating-direction implicit (ADI) method, equation (19.3.16), is an example of operator splitting with a slightly different twist. Let us reinterpret (19.3.19) to have a different meaning: Let $\mathcal{U}_1$ now denote an updating method that includes algebraically *all* the pieces of the total operator $\mathcal{L}$, but which is desirably *stable* only for the $\mathcal{L}_1$ piece; likewise $\mathcal{U}_2, \ldots \mathcal{U}_m$. Then a method of getting from u^n to u^{n+1} is

$$\begin{aligned} u^{n+1/m} &= \mathcal{U}_1(u^n, \Delta t/m) \\ u^{n+2/m} &= \mathcal{U}_2(u^{n+1/m}, \Delta t/m) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t/m) \end{aligned} \tag{19.3.22}$$

The timestep for each fractional step in (19.3.22) is now only $1/m$ of the full timestep, because each partial operation acts with all the terms of the original operator.

Equation (19.3.22) is usually, though not always, stable as a differencing scheme for the operator $\mathcal{L}$. In fact, as a rule of thumb, it is often sufficient to have stable $\mathcal{U}_i$'s only for the operator pieces having the highest number of spatial derivatives — the other $\mathcal{U}_i$'s can be *unstable* — to make the overall scheme stable!

It is at this point that we turn our attention from initial value problems to boundary value problems. These will occupy us for the remainder of the chapter.

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press).

19.4 Fourier and Cyclic Reduction Methods for Boundary Value Problems

As discussed in §19.0, most boundary value problems (elliptic equations, for example) reduce to solving large sparse linear systems of the form

$$\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.4.1}$$

either once, for boundary value equations that are linear, or iteratively, for boundary value equations that are nonlinear.

For example, a combined advective-diffusion equation, such as

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + D\frac{\partial^2 u}{\partial x^2} \tag{19.3.21}$$

might profitably use an explicit scheme for the advective term combined with a Crank-Nicholson or other implicit scheme for the diffusion term.

The alternating-direction implicit (ADI) method, equation (19.3.16), is an example of operator splitting with a slightly different twist. Let us reinterpret (19.3.19) to have a different meaning: Let $\mathcal{U}_1$ now denote an updating method that includes algebraically *all* the pieces of the total operator $\mathcal{L}$, but which is desirably *stable* only for the $\mathcal{L}_1$ piece; likewise $\mathcal{U}_2, \ldots \mathcal{U}_m$. Then a method of getting from u^n to u^{n+1} is

$$\begin{aligned} u^{n+1/m} &= \mathcal{U}_1(u^n, \Delta t/m) \\ u^{n+2/m} &= \mathcal{U}_2(u^{n+1/m}, \Delta t/m) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t/m) \end{aligned} \tag{19.3.22}$$

The timestep for each fractional step in (19.3.22) is now only $1/m$ of the full timestep, because each partial operation acts with all the terms of the original operator.

Equation (19.3.22) is usually, though not always, stable as a differencing scheme for the operator $\mathcal{L}$. In fact, as a rule of thumb, it is often sufficient to have stable $\mathcal{U}_i$'s only for the operator pieces having the highest number of spatial derivatives — the other $\mathcal{U}_i$'s can be *unstable* — to make the overall scheme stable!

It is at this point that we turn our attention from initial value problems to boundary value problems. These will occupy us for the remainder of the chapter.

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press).

19.4 Fourier and Cyclic Reduction Methods for Boundary Value Problems

As discussed in §19.0, most boundary value problems (elliptic equations, for example) reduce to solving large sparse linear systems of the form

$$\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.4.1}$$

either once, for boundary value equations that are linear, or iteratively, for boundary value equations that are nonlinear.

Two important techniques lead to "rapid" solution of equation (19.4.1) when the sparse matrix is of certain frequently occurring forms. The *Fourier transform method* is directly applicable when the equations have coefficients that are constant in space. The *cyclic reduction* method is somewhat more general; its applicability is related to the question of whether the equations are separable (in the sense of "separation of variables"). Both methods require the boundaries to coincide with the coordinate lines. Finally, for some problems, there is a powerful combination of these two methods called *FACR (Fourier Analysis and Cyclic Reduction)*. We now consider each method in turn, using equation (19.0.3), with finite-difference representation (19.0.6), as a model example. Generally speaking, the methods in this section are faster, when they apply, than the simpler relaxation methods discussed in §19.5; but they are not necessarily faster than the more complicated multigrid methods discussed in §19.6.

Fourier Transform Method

The discrete inverse Fourier transform in both x and y is

$$u_{jl} = \frac{1}{JL} \sum_{m=0}^{J-1} \sum_{n=0}^{L-1} \widehat{u}_{mn} e^{-2\pi i j m/J} e^{-2\pi i l n/L} \tag{19.4.2}$$

This can be computed using the FFT independently in each dimension, or else all at once via the routine `fourn` of §12.4 or the routine `rlft3` of §12.5. Similarly,

$$\rho_{jl} = \frac{1}{JL} \sum_{m=0}^{J-1} \sum_{n=0}^{L-1} \widehat{\rho}_{mn} e^{-2\pi i j m/J} e^{-2\pi i l n/L} \tag{19.4.3}$$

If we substitute expressions (19.4.2) and (19.4.3) in our model problem (19.0.6), we find

$$\widehat{u}_{mn} \left(e^{2\pi i m/J} + e^{-2\pi i m/J} + e^{2\pi i n/L} + e^{-2\pi i n/L} - 4 \right) = \widehat{\rho}_{mn} \Delta^2 \tag{19.4.4}$$

or

$$\widehat{u}_{mn} = \frac{\widehat{\rho}_{mn} \Delta^2}{2 \left(\cos \dfrac{2\pi m}{J} + \cos \dfrac{2\pi n}{L} - 2 \right)} \tag{19.4.5}$$

Thus the strategy for solving equation (19.0.6) by FFT techniques is:

- Compute $\widehat{\rho}_{mn}$ as the Fourier transform

$$\widehat{\rho}_{mn} = \sum_{j=0}^{J-1} \sum_{l=0}^{L-1} \rho_{jl}\, e^{2\pi i m j/J} e^{2\pi i n l/L} \tag{19.4.6}$$

- Compute $\widehat{u}_{mn}$ from equation (19.4.5).

- Compute u_{jl} by the inverse Fourier transform (19.4.2).

The above procedure is valid for periodic boundary conditions. In other words, the solution satisfies

$$u_{jl} = u_{j+J,l} = u_{j,l+L} \tag{19.4.7}$$

Next consider a Dirichlet boundary condition $u = 0$ on the rectangular boundary. Instead of the expansion (19.4.2), we now need an expansion in sine waves:

$$u_{jl} = \frac{2}{J}\frac{2}{L}\sum_{m=1}^{J-1}\sum_{n=1}^{L-1}\widehat{u}_{mn}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.8}$$

This satisfies the boundary conditions that $u = 0$ at $j = 0, J$ and at $l = 0, L$. If we substitute this expansion and the analogous one for ρ_{jl} into equation (19.0.6), we find that the solution procedure parallels that for periodic boundary conditions:

- Compute $\widehat{\rho}_{mn}$ by the sine transform

$$\widehat{\rho}_{mn} = \sum_{j=1}^{J-1}\sum_{l=1}^{L-1}\rho_{jl}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.9}$$

(A fast sine transform algorithm was given in §12.3.)

- Compute $\widehat{u}_{mn}$ from the expression analogous to (19.4.5),

$$\widehat{u}_{mn} = \frac{\Delta^2\widehat{\rho}_{mn}}{2\left(\cos\dfrac{\pi m}{J} + \cos\dfrac{\pi n}{L} - 2\right)} \tag{19.4.10}$$

- Compute u_{jl} by the inverse sine transform (19.4.8).

If we have inhomogeneous boundary conditions, for example $u = 0$ on all boundaries except $u = f(y)$ on the boundary $x = J\Delta$, we have to add to the above solution a solution u^H of the homogeneous equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{19.4.11}$$

that satisfies the required boundary conditions. In the continuum case, this would be an expression of the form

$$u^H = \sum_n A_n \sinh\frac{n\pi x}{J\Delta}\sin\frac{n\pi y}{L\Delta} \tag{19.4.12}$$

where A_n would be found by requiring that $u = f(y)$ at $x = J\Delta$. In the discrete case, we have

$$u^H_{jl} = \frac{2}{L}\sum_{n=1}^{L-1} A_n \sinh\frac{\pi nj}{J}\sin\frac{\pi nl}{L} \tag{19.4.13}$$

If $f(y = l\Delta) \equiv f_l$, then we get A_n from the inverse formula

$$A_n = \frac{1}{\sinh \pi n} \sum_{l=1}^{L-1} f_l \sin \frac{\pi n l}{L} \tag{19.4.14}$$

The complete solution to the problem is

$$u = u_{jl} + u_{jl}^H \tag{19.4.15}$$

By adding appropriate terms of the form (19.4.12), we can handle inhomogeneous terms on any boundary surface.

A much simpler procedure for handling inhomogeneous terms is to note that whenever boundary terms appear on the left-hand side of (19.0.6), they can be taken over to the right-hand side since they are known. The effective source term is therefore ρ_{jl} plus a contribution from the boundary terms. To implement this idea formally, write the solution as

$$u = u' + u^B \tag{19.4.16}$$

where $u' = 0$ on the boundary, while u^B vanishes everywhere *except* on the boundary. There it takes on the given boundary value. In the above example, the only nonzero values of u^B would be

$$u^B_{J,l} = f_l \tag{19.4.17}$$

The model equation (19.0.3) becomes

$$\nabla^2 u' = -\nabla^2 u^B + \rho \tag{19.4.18}$$

or, in finite-difference form,

$$\begin{aligned} u'_{j+1,l} + u'_{j-1,l} + u'_{j,l+1} + u'_{j,l-1} - 4u'_{j,l} = \\ -(u^B_{j+1,l} + u^B_{j-1,l} + u^B_{j,l+1} + u^B_{j,l-1} - 4u^B_{j,l}) + \Delta^2 \rho_{j,l} \end{aligned} \tag{19.4.19}$$

All the u^B terms in equation (19.4.19) vanish except when the equation is evaluated at $j = J - 1$, where

$$u'_{J,l} + u'_{J-2,l} + u'_{J-1,l+1} + u'_{J-1,l-1} - 4u'_{J-1,l} = -f_l + \Delta^2 \rho_{J-1,l} \tag{19.4.20}$$

Thus the problem is now equivalent to the case of zero boundary conditions, except that one row of the source term is modified by the replacement

$$\Delta^2 \rho_{J-1,l} \to \Delta^2 \rho_{J-1,l} - f_l \tag{19.4.21}$$

The case of Neumann boundary conditions $\nabla u = 0$ is handled by the cosine expansion (12.3.17):

$$u_{jl} = \frac{2}{J}\frac{2}{L} \sum_{m=0}^{J}{}'' \sum_{n=0}^{L}{}'' \widehat{u}_{mn} \cos \frac{\pi j m}{J} \cos \frac{\pi l n}{L} \tag{19.4.22}$$

Here the double prime notation means that the terms for $m = 0$ and $m = J$ should be multiplied by $\frac{1}{2}$, and similarly for $n = 0$ and $n = L$. Inhomogeneous terms $\nabla u = g$ can be again included by adding a suitable solution of the homogeneous equation, or more simply by taking boundary terms over to the right-hand side. For example, the condition

$$\frac{\partial u}{\partial x} = g(y) \qquad \text{at} \quad x = 0 \tag{19.4.23}$$

becomes

$$\frac{u_{1,l} - u_{-1,l}}{2\Delta} = g_l \tag{19.4.24}$$

where $g_l \equiv g(y = l\Delta)$. Once again we write the solution in the form (19.4.16), where now $\nabla u' = 0$ on the boundary. This time ∇u^B takes on the prescribed value on the boundary, but u^B vanishes everywhere except just *outside* the boundary. Thus equation (19.4.24) gives

$$u^B_{-1,l} = -2\Delta g_l \tag{19.4.25}$$

All the u^B terms in equation (19.4.19) vanish except when $j = 0$:

$$u'_{1,l} + u'_{-1,l} + u'_{0,l+1} + u'_{0,l-1} - 4u'_{0,l} = 2\Delta g_l + \Delta^2 \rho_{0,l} \tag{19.4.26}$$

Thus u' is the solution of a zero-gradient problem, with the source term modified by the replacement

$$\Delta^2 \rho_{0,l} \to \Delta^2 \rho_{0,l} + 2\Delta g_l \tag{19.4.27}$$

Sometimes Neumann boundary conditions are handled by using a staggered grid, with the u's defined midway between zone boundaries so that first derivatives are centered on the mesh points. You can solve such problems using similar techniques to those described above if you use the alternative form of the cosine transform, equation (12.3.23).

Cyclic Reduction

Evidently the FFT method works only when the original PDE has constant coefficients, and boundaries that coincide with the coordinate lines. An alternative algorithm, which can be used on somewhat more general equations, is called *cyclic reduction (CR)*.

We illustrate cyclic reduction on the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + b(y)\frac{\partial u}{\partial y} + c(y)u = g(x, y) \tag{19.4.28}$$

This form arises very often in practice from the Helmholtz or Poisson equations in polar, cylindrical, or spherical coordinate systems. More general separable equations are treated in [1].

The finite-difference form of equation (19.4.28) can be written as a set of vector equations

$$\mathbf{u}_{j-1} + \mathbf{T} \cdot \mathbf{u}_j + \mathbf{u}_{j+1} = \mathbf{g}_j \Delta^2 \tag{19.4.29}$$

Here the index j comes from differencing in the x-direction, while the y-differencing (denoted by the index l previously) has been left in vector form. The matrix $\mathbf{T}$ has the form

$$\mathbf{T} = \mathbf{B} - 2\mathbf{1} \tag{19.4.30}$$

where the $2\mathbf{1}$ comes from the x-differencing and the matrix $\mathbf{B}$ from the y-differencing. The matrix $\mathbf{B}$, and hence $\mathbf{T}$, is tridiagonal with variable coefficients.

The CR method is derived by writing down three successive equations like (19.4.29):

$$\begin{aligned} \mathbf{u}_{j-2} + \mathbf{T} \cdot \mathbf{u}_{j-1} + \mathbf{u}_j &= \mathbf{g}_{j-1} \Delta^2 \\ \mathbf{u}_{j-1} + \mathbf{T} \cdot \mathbf{u}_j + \mathbf{u}_{j+1} &= \mathbf{g}_j \Delta^2 \\ \mathbf{u}_j + \mathbf{T} \cdot \mathbf{u}_{j+1} + \mathbf{u}_{j+2} &= \mathbf{g}_{j+1} \Delta^2 \end{aligned} \tag{19.4.31}$$

Matrix-multiplying the middle equation by $-\mathbf{T}$ and then adding the three equations, we get

$$\mathbf{u}_{j-2} + \mathbf{T}^{(1)} \cdot \mathbf{u}_j + \mathbf{u}_{j+2} = \mathbf{g}_j^{(1)} \Delta^2 \tag{19.4.32}$$

This is an equation of the same form as (19.4.29), with

$$\begin{aligned} \mathbf{T}^{(1)} &= 2\mathbf{1} - \mathbf{T}^2 \\ \mathbf{g}_j^{(1)} &= \Delta^2 (\mathbf{g}_{j-1} - \mathbf{T} \cdot \mathbf{g}_j + \mathbf{g}_{j+1}) \end{aligned} \tag{19.4.33}$$

After one level of CR, we have reduced the number of equations by a factor of two. Since the resulting equations are of the same form as the original equation, we can repeat the process. Taking the number of mesh points to be a power of 2 for simplicity, we finally end up with a single equation for the central line of variables:

$$\mathbf{T}^{(f)} \cdot \mathbf{u}_{J/2} = \Delta^2 \mathbf{g}_{J/2}^{(f)} - \mathbf{u}_0 - \mathbf{u}_J \tag{19.4.34}$$

Here we have moved $\mathbf{u}_0$ and $\mathbf{u}_J$ to the right-hand side because they are known boundary values. Equation (19.4.34) can be solved for $\mathbf{u}_{J/2}$ by the standard tridiagonal algorithm. The two equations at level $f-1$ involve $\mathbf{u}_{J/4}$ and $\mathbf{u}_{3J/4}$. The equation for $\mathbf{u}_{J/4}$ involves $\mathbf{u}_0$ and $\mathbf{u}_{J/2}$, both of which are known, and hence can be solved by the usual tridiagonal routine. A similar result holds true at every stage, so we end up solving $J-1$ tridiagonal systems.

In practice, equations (19.4.33) should be rewritten to avoid numerical instability. For these and other practical details, refer to [2].

FACR Method

The *best* way to solve equations of the form (19.4.28), including the constant coefficient problem (19.0.3), is a combination of Fourier analysis and cyclic reduction, the FACR method [3-6]. If at the rth stage of CR we Fourier analyze the equations of the form (19.4.32) along y, that is, with respect to the suppressed vector index, we will have a tridiagonal system in the x-direction for each y-Fourier mode:

$$\widehat{u}^k_{j-2^r} + \lambda^{(r)}_k \widehat{u}^k_j + \widehat{u}^k_{j+2^r} = \Delta^2 g^{(r)k}_j \tag{19.4.35}$$

Here $\lambda^{(r)}_k$ is the eigenvalue of $\mathbf{T}^{(r)}$ corresponding to the kth Fourier mode. For the equation (19.0.3), equation (19.4.5) shows that $\lambda^{(r)}_k$ will involve terms like $\cos(2\pi k/L) - 2$ raised to a power. Solve the tridiagonal systems for $\widehat{u}^k_j$ at the levels $j = 2^r, 2 \times 2^r, 4 \times 2^r, ..., J - 2^r$. Fourier synthesize to get the y-values on these x-lines. Then fill in the intermediate x-lines as in the original CR algorithm.

The trick is to choose the number of levels of CR so as to minimize the total number of arithmetic operations. One can show that for a typical case of a 128×128 mesh, the optimal level is $r = 2$; asymptotically, $r \to \log_2(\log_2 J)$.

A rough estimate of running times for these algorithms for equation (19.0.3) is as follows: The FFT method (in both x and y) and the CR method are roughly comparable. FACR with $r = 0$ (that is, FFT in one dimension and solve the tridiagonal equations by the usual algorithm in the other dimension) gives about a factor of two gain in speed. The optimal FACR with $r = 2$ gives another factor of two gain in speed.

CITED REFERENCES AND FURTHER READING:

Swartzrauber, P.N. 1977, *SIAM Review*, vol. 19, pp. 490–501. [1]

Buzbee, B.L, Golub, G.H., and Nielson, C.W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 627–656; see also *op. cit.* vol. 11, pp. 753–763. [2]

Hockney, R.W. 1965, *Journal of the Association for Computing Machinery*, vol. 12, pp. 95–113. [3]

Hockney, R.W. 1970, in *Methods of Computational Physics*, vol. 9 (New York: Academic Press), pp. 135–211. [4]

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6. [5]

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329. [6]

19.5 Relaxation Methods for Boundary Value Problems

As we mentioned in §19.0, relaxation methods involve splitting the sparse matrix that arises from finite differencing and then iterating until a solution is found.

There is another way of thinking about relaxation methods that is somewhat more physical. Suppose we wish to solve the elliptic equation

$$\mathcal{L}u = \rho \tag{19.5.1}$$

FACR Method

The *best* way to solve equations of the form (19.4.28), including the constant coefficient problem (19.0.3), is a combination of Fourier analysis and cyclic reduction, the FACR method [3-6]. If at the rth stage of CR we Fourier analyze the equations of the form (19.4.32) along y, that is, with respect to the suppressed vector index, we will have a tridiagonal system in the x-direction for each y-Fourier mode:

$$\widehat{u}^k_{j-2^r} + \lambda^{(r)}_k \widehat{u}^k_j + \widehat{u}^k_{j+2^r} = \Delta^2 g^{(r)k}_j \tag{19.4.35}$$

Here $\lambda^{(r)}_k$ is the eigenvalue of $\mathbf{T}^{(r)}$ corresponding to the kth Fourier mode. For the equation (19.0.3), equation (19.4.5) shows that $\lambda^{(r)}_k$ will involve terms like $\cos(2\pi k/L) - 2$ raised to a power. Solve the tridiagonal systems for $\widehat{u}^k_j$ at the levels $j = 2^r, 2 \times 2^r, 4 \times 2^r, ..., J - 2^r$. Fourier synthesize to get the y-values on these x-lines. Then fill in the intermediate x-lines as in the original CR algorithm.

The trick is to choose the number of levels of CR so as to minimize the total number of arithmetic operations. One can show that for a typical case of a 128×128 mesh, the optimal level is $r = 2$; asymptotically, $r \to \log_2(\log_2 J)$.

A rough estimate of running times for these algorithms for equation (19.0.3) is as follows: The FFT method (in both x and y) and the CR method are roughly comparable. FACR with $r = 0$ (that is, FFT in one dimension and solve the tridiagonal equations by the usual algorithm in the other dimension) gives about a factor of two gain in speed. The optimal FACR with $r = 2$ gives another factor of two gain in speed.

CITED REFERENCES AND FURTHER READING:

Swartzrauber, P.N. 1977, *SIAM Review*, vol. 19, pp. 490–501. [1]

Buzbee, B.L, Golub, G.H., and Nielson, C.W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 627–656; see also *op. cit.* vol. 11, pp. 753–763. [2]

Hockney, R.W. 1965, *Journal of the Association for Computing Machinery*, vol. 12, pp. 95–113. [3]

Hockney, R.W. 1970, in *Methods of Computational Physics*, vol. 9 (New York: Academic Press), pp. 135–211. [4]

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6. [5]

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329. [6]

19.5 Relaxation Methods for Boundary Value Problems

As we mentioned in §19.0, relaxation methods involve splitting the sparse matrix that arises from finite differencing and then iterating until a solution is found.

There is another way of thinking about relaxation methods that is somewhat more physical. Suppose we wish to solve the elliptic equation

$$\mathcal{L}u = \rho \tag{19.5.1}$$

where $\mathcal{L}$ represents some elliptic operator and ρ is the source term. Rewrite the equation as a diffusion equation,

$$\frac{\partial u}{\partial t} = \mathcal{L}u - \rho \tag{19.5.2}$$

An initial distribution u *relaxes* to an equilibrium solution as $t \to \infty$. This equilibrium has all time derivatives vanishing. Therefore it is the solution of the original elliptic problem (19.5.1). We see that all the machinery of §19.2, on diffusive initial value equations, can be brought to bear on the solution of boundary value problems by relaxation methods.

Let us apply this idea to our model problem (19.0.3). The diffusion equation is

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - \rho \tag{19.5.3}$$

If we use FTCS differencing (cf. equation 19.2.4), we get

$$u_{j,l}^{n+1} = u_{j,l}^n + \frac{\Delta t}{\Delta^2}\left(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n - 4u_{j,l}^n\right) - \rho_{j,l}\Delta t \tag{19.5.4}$$

Recall from (19.2.6) that FTCS differencing is stable in one spatial dimension only if $\Delta t/\Delta^2 \le \frac{1}{2}$. In two dimensions this becomes $\Delta t/\Delta^2 \le \frac{1}{4}$. Suppose we try to take the largest possible timestep, and set $\Delta t = \Delta^2/4$. Then equation (19.5.4) becomes

$$u_{j,l}^{n+1} = \frac{1}{4}\left(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n\right) - \frac{\Delta^2}{4}\rho_{j,l} \tag{19.5.5}$$

Thus the algorithm consists of using the average of u at its four nearest-neighbor points on the grid (plus the contribution from the source). This procedure is then iterated until convergence.

This method is in fact a classical method with origins dating back to the last century, called *Jacobi's method* (not to be confused with the Jacobi method for eigenvalues). The method is not practical because it converges too slowly. However, it is the basis for understanding the modern methods, which are always compared with it.

Another classical method is the *Gauss-Seidel* method, which turns out to be important in multigrid methods (§19.6). Here we make use of updated values of u on the right-hand side of (19.5.5) as soon as they become available. In other words, the averaging is done "in place" instead of being "copied" from an earlier timestep to a later one. If we are proceeding along the rows, incrementing j for fixed l, we have

$$u_{j,l}^{n+1} = \frac{1}{4}\left(u_{j+1,l}^n + u_{j-1,l}^{n+1} + u_{j,l+1}^n + u_{j,l-1}^{n+1}\right) - \frac{\Delta^2}{4}\rho_{j,l} \tag{19.5.6}$$

This method is also slowly converging and only of theoretical interest when used by itself, but some analysis of it will be instructive.

Let us look at the Jacobi and Gauss-Seidel methods in terms of the matrix splitting concept. We change notation and call **u** "**x**," to conform to standard matrix notation. To solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{19.5.7}$$

we can consider splitting $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U} \tag{19.5.8}$$

where $\mathbf{D}$ is the diagonal part of $\mathbf{A}$, $\mathbf{L}$ is the lower triangle of $\mathbf{A}$ with zeros on the diagonal, and $\mathbf{U}$ is the upper triangle of $\mathbf{A}$ with zeros on the diagonal.

In the Jacobi method we write for the rth step of iteration

$$\mathbf{D} \cdot \mathbf{x}^{(r)} = -(\mathbf{L} + \mathbf{U}) \cdot \mathbf{x}^{(r-1)} + \mathbf{b} \tag{19.5.9}$$

For our model problem (19.5.5), $\mathbf{D}$ is simply the identity matrix. The Jacobi method converges for matrices $\mathbf{A}$ that are "diagonally dominant" in a sense that can be made mathematically precise. For matrices arising from finite differencing, this condition is usually met.

What is the rate of convergence of the Jacobi method? A detailed analysis is beyond our scope, but here is some of the flavor: The matrix $-\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U})$ is the *iteration matrix* which, apart from an additive term, maps one set of $\mathbf{x}$'s into the next. The iteration matrix has eigenvalues, each one of which reflects the factor by which the amplitude of a particular eigenmode of undesired residual is suppressed during one iteration. Evidently those factors had better all have modulus < 1 for the relaxation to work at all! The rate of convergence of the method is set by the rate for the slowest-decaying eigenmode, i.e., the factor with largest modulus. The modulus of this largest factor, therefore lying between 0 and 1, is called the *spectral radius* of the relaxation operator, denoted ρ_s.

The number of iterations r required to reduce the overall error by a factor 10^{-p} is thus estimated by

$$r \approx \frac{p \ln 10}{(-\ln \rho_s)} \tag{19.5.10}$$

In general, the spectral radius ρ_s goes asymptotically to the value 1 as the grid size J is increased, so that more iterations are required. For any given equation, grid geometry, *and boundary condition*, the spectral radius can, in principle, be computed analytically. For example, for equation (19.5.5) on a $J \times J$ grid with Dirichlet boundary conditions on all four sides, the asymptotic formula for large J turns out to be

$$\rho_s \simeq 1 - \frac{\pi^2}{2J^2} \tag{19.5.11}$$

The number of iterations r required to reduce the error by a factor of 10^{-p} is thus

$$r \simeq \frac{2pJ^2 \ln 10}{\pi^2} \simeq \frac{1}{2} pJ^2 \tag{19.5.12}$$

In other words, the number of iterations is proportional to the number of mesh points, J^2. Since 100×100 and larger problems are common, it is clear that the Jacobi method is only of theoretical interest.

The Gauss-Seidel method, equation (19.5.6), corresponds to the matrix decomposition

$$(\mathbf{L}+\mathbf{D})\cdot\mathbf{x}^{(r)}=-\mathbf{U}\cdot\mathbf{x}^{(r-1)}+\mathbf{b} \tag{19.5.13}$$

The fact that $\mathbf{L}$ is on the left-hand side of the equation follows from the updating in place, as you can easily check if you write out (19.5.13) in components. One can show [1-3] that the spectral radius is just the square of the spectral radius of the Jacobi method. For our model problem, therefore,

$$\rho_s\simeq 1-\frac{\pi^2}{J^2} \tag{19.5.14}$$

$$r\simeq\frac{pJ^2\ln 10}{\pi^2}\simeq\frac{1}{4}pJ^2 \tag{19.5.15}$$

The factor of two improvement in the number of iterations over the Jacobi method still leaves the method impractical.

Successive Overrelaxation (SOR)

We get a better algorithm — one that was the standard algorithm until the 1970s — if we make an *overcorrection* to the value of $\mathbf{x}^{(r)}$ at the rth stage of Gauss-Seidel iteration, thus anticipating future corrections. Solve (19.5.13) for $\mathbf{x}^{(r)}$, add and subtract $\mathbf{x}^{(r-1)}$ on the right-hand side, and hence write the Gauss-Seidel method as

$$\mathbf{x}^{(r)}=\mathbf{x}^{(r-1)}-(\mathbf{L}+\mathbf{D})^{-1}\cdot[(\mathbf{L}+\mathbf{D}+\mathbf{U})\cdot\mathbf{x}^{(r-1)}-\mathbf{b}] \tag{19.5.16}$$

The term in square brackets is just the residual vector $\xi^{(r-1)}$, so

$$\mathbf{x}^{(r)}=\mathbf{x}^{(r-1)}-(\mathbf{L}+\mathbf{D})^{-1}\cdot\xi^{(r-1)} \tag{19.5.17}$$

Now *overcorrect*, defining

$$\mathbf{x}^{(r)}=\mathbf{x}^{(r-1)}-\omega(\mathbf{L}+\mathbf{D})^{-1}\cdot\xi^{(r-1)} \tag{19.5.18}$$

Here ω is called the *overrelaxation parameter*, and the method is called *successive overrelaxation* (SOR).

The following theorems can be proved [1-3]:

- The method is convergent only for $0<\omega<2$. If $0<\omega<1$, we speak of *underrelaxation*.
- Under certain mathematical restrictions generally satisfied by matrices arising from finite differencing, only overrelaxation ($1<\omega<2$) can give faster convergence than the Gauss-Seidel method.
- If $\rho_{\rm Jacobi}$ is the spectral radius of the Jacobi iteration (so that the square of it is the spectral radius of the Gauss-Seidel iteration), then the *optimal* choice for ω is given by

$$\omega=\frac{2}{1+\sqrt{1-\rho_{\rm Jacobi}^2}} \tag{19.5.19}$$

- For this optimal choice, the spectral radius for SOR is

$$\rho_{\rm SOR} = \left(\frac{\rho_{\rm Jacobi}}{1+\sqrt{1-\rho_{\rm Jacobi}^2}} \right)^2 \tag{19.5.20}$$

As an application of the above results, consider our model problem for which $\rho_{\rm Jacobi}$ is given by equation (19.5.11). Then equations (19.5.19) and (19.5.20) give

$$\omega \simeq \frac{2}{1+\pi/J} \tag{19.5.21}$$

$$\rho_{\rm SOR} \simeq 1 - \frac{2\pi}{J} \qquad \text{for large} \quad J \tag{19.5.22}$$

Equation (19.5.10) gives for the number of iterations to reduce the initial error by a factor of 10^{-p},

$$r \simeq \frac{pJ\ln 10}{2\pi} \simeq \frac{1}{3}pJ \tag{19.5.23}$$

Comparing with equation (19.5.12) or (19.5.15), we see that optimal SOR requires of order J iterations, as opposed to of order J^2. Since J is typically 100 or larger, this makes a tremendous difference! Equation (19.5.23) leads to the mnemonic that 3-figure accuracy ($p = 3$) requires a number of iterations equal to the number of mesh points along a side of the grid. For 6-figure accuracy, we require about twice as many iterations.

How do we choose ω for a problem for which the answer is not known analytically? That is just the weak point of SOR! The advantages of SOR obtain only in a fairly narrow window around the correct value of ω. It is better to take ω slightly too large, rather than slightly too small, but best to get it right.

One way to choose ω is to map your problem approximately onto a known problem, replacing the coefficients in the equation by average values. Note, however, that the known problem must have the same grid size and boundary conditions as the actual problem. We give for reference purposes the value of $\rho_{\rm Jacobi}$ for our model problem on a rectangular $J \times L$ grid, allowing for the possibility that $\Delta x \neq \Delta y$:

$$\rho_{\rm Jacobi} = \frac{\cos\dfrac{\pi}{J} + \left(\dfrac{\Delta x}{\Delta y}\right)^2 \cos\dfrac{\pi}{L}}{1+\left(\dfrac{\Delta x}{\Delta y}\right)^2} \tag{19.5.24}$$

Equation (19.5.24) holds for homogeneous Dirichlet or Neumann boundary conditions. For periodic boundary conditions, make the replacement $\pi \to 2\pi$.

A second way, which is especially useful if you plan to solve many similar elliptic equations each time with slightly different coefficients, is to determine the optimum value ω empirically on the first equation and then use that value for the remaining equations. Various automated schemes for doing this and for "seeking out" the best values of ω are described in the literature.

While the matrix notation introduced earlier is useful for theoretical analyses, for practical implementation of the SOR algorithm we need explicit formulas.

Consider a general second-order elliptic equation in x and y, finite differenced on a square as for our model equation. Corresponding to each row of the matrix **A** is an equation of the form

$$a_{j,l}u_{j+1,l} + b_{j,l}u_{j-1,l} + c_{j,l}u_{j,l+1} + d_{j,l}u_{j,l-1} + e_{j,l}u_{j,l} = f_{j,l} \tag{19.5.25}$$

For our model equation, we had $a = b = c = d = 1, e = -4$. The quantity f is proportional to the source term. The iterative procedure is defined by solving (19.5.25) for $u_{j,l}$:

$$u^*_{j,l} = \frac{1}{e_{j,l}}\left(f_{j,l} - a_{j,l}u_{j+1,l} - b_{j,l}u_{j-1,l} - c_{j,l}u_{j,l+1} - d_{j,l}u_{j,l-1}\right) \tag{19.5.26}$$

Then $u^{\rm new}_{j,l}$ is a weighted average

$$u^{\rm new}_{j,l} = \omega u^*_{j,l} + (1-\omega)u^{\rm old}_{j,l} \tag{19.5.27}$$

We calculate it as follows: The residual at any stage is

$$\xi_{j,l} = a_{j,l}u_{j+1,l} + b_{j,l}u_{j-1,l} + c_{j,l}u_{j,l+1} + d_{j,l}u_{j,l-1} + e_{j,l}u_{j,l} - f_{j,l} \tag{19.5.28}$$

and the SOR algorithm (19.5.18) or (19.5.27) is

$$u^{\rm new}_{j,l} = u^{\rm old}_{j,l} - \omega\frac{\xi_{j,l}}{e_{j,l}} \tag{19.5.29}$$

This formulation is very easy to program, and the norm of the residual vector $\xi_{j,l}$ can be used as a criterion for terminating the iteration.

Another practical point concerns the order in which mesh points are processed. The obvious strategy is simply to proceed in order down the rows (or columns). Alternatively, suppose we divide the mesh into "odd" and "even" meshes, like the red and black squares of a checkerboard. Then equation (19.5.26) shows that the odd points depend only on the even mesh values and vice versa. Accordingly, we can carry out one half-sweep updating the odd points, say, and then another half-sweep updating the even points with the new odd values. For the version of SOR implemented below, we shall adopt odd-even ordering.

The last practical point is that in practice the asymptotic rate of convergence in SOR is not attained until of order J iterations. The error often grows by a factor of 20 before convergence sets in. A trivial modification to SOR resolves this problem. It is based on the observation that, while ω is the optimum *asymptotic* relaxation parameter, it is not necessarily a good initial choice. In SOR with *Chebyshev acceleration*, one uses odd-even ordering and changes ω at each half-sweep according to the following prescription:

$$\begin{aligned}
\omega^{(0)} &= 1 \\
\omega^{(1/2)} &= 1/(1-\rho^2_{\rm Jacobi}/2) \\
\omega^{(n+1/2)} &= 1/(1-\rho^2_{\rm Jacobi}\omega^{(n)}/4), \qquad n = 1/2, 1, ..., \infty \\
\omega^{(\infty)} &\to \omega_{\rm optimal}
\end{aligned} \tag{19.5.30}$$

The beauty of Chebyshev acceleration is that the norm of the error always decreases with each iteration. (This is the norm of the actual error in $u_{j,l}$. The norm of the residual $\xi_{j,l}$ need not decrease monotonically.) While the asymptotic rate of convergence is the same as ordinary SOR, there is never any excuse for not using Chebyshev acceleration to reduce the total number of iterations required.

Here we give a routine for SOR with Chebyshev acceleration.

```
      SUBROUTINE sor(a,b,c,d,e,f,u,jmax,rjac)
      INTEGER jmax,MAXITS
      DOUBLE PRECISION rjac,a(jmax,jmax),b(jmax,jmax),
     *     c(jmax,jmax),d(jmax,jmax),e(jmax,jmax),
     *     f(jmax,jmax),u(jmax,jmax),EPS
      PARAMETER (MAXITS=1000,EPS=1.d-5)
```

Successive overrelaxation solution of equation (19.5.25) with Chebyshev acceleration. a, b, c, d, e, and f are input as the coefficients of the equation, each dimensioned to the grid size JMAX × JMAX. u is input as the initial guess to the solution, usually zero, and returns with the final value. rjac is input as the spectral radius of the Jacobi iteration, or an estimate of it.

```
      INTEGER ipass,j,jsw,l,lsw,n
      DOUBLE PRECISION anorm,anormf,
     *     omega,resid          Double precision is a good idea for JMAX bigger than about 25.
      anormf=0.d0               Compute initial norm of residual and terminate iteration when
      do 12 j=2,jmax-1              norm has been reduced by a factor EPS.
        do 11 l=2,jmax-1
          anormf=anormf+abs(f(j,l))       Assumes initial u is zero.
        enddo 11
      enddo 12
      omega=1.d0
      do 16 n=1,MAXITS
        anorm=0.d0
        jsw=1
        do 15 ipass=1,2                   Odd-even ordering.
          lsw=jsw
          do 14 j=2,jmax-1
            do 13 l=lsw+1,jmax-1,2
              resid=a(j,l)*u(j+1,l)+b(j,l)*u(j-1,l)+
     *                c(j,l)*u(j,l+1)+d(j,l)*u(j,l-1)+
     *                e(j,l)*u(j,l)-f(j,l)
              anorm=anorm+abs(resid)
              u(j,l)=u(j,l)-omega*resid/e(j,l)
            enddo 13
            lsw=3-lsw
          enddo 14
          jsw=3-jsw
          if(n.eq.1.and.ipass.eq.1) then
            omega=1.d0/(1.d0-.5d0*rjac**2)
          else
            omega=1.d0/(1.d0-.25d0*rjac**2*omega)
          endif
        enddo 15
        if(anorm.lt.EPS*anormf)return
      enddo 16
      pause 'MAXITS exceeded in sor'
      END
```

The main advantage of SOR is that it is very easy to program. Its main disadvantage is that it is still very inefficient on large problems.

ADI (Alternating-Direction Implicit) Method

The ADI method of §19.3 for diffusion equations can be turned into a relaxation method for elliptic equations [1-4]. In §19.3, we discussed ADI as a method for solving the time-dependent heat-flow equation

$$\frac{\partial u}{\partial t} = \nabla^2 u - \rho \tag{19.5.31}$$

By letting $t \to \infty$ one also gets an iterative method for solving the elliptic equation

$$\nabla^2 u = \rho \tag{19.5.32}$$

In either case, the operator splitting is of the form

$$\mathcal{L} = \mathcal{L}_x + \mathcal{L}_y \tag{19.5.33}$$

where $\mathcal{L}_x$ represents the differencing in x and $\mathcal{L}_y$ that in y.

For example, in our model problem (19.0.6) with $\Delta x = \Delta y = \Delta$, we have

$$\begin{aligned} \mathcal{L}_x u &= 2u_{j,l} - u_{j+1,l} - u_{j-1,l} \\ \mathcal{L}_y u &= 2u_{j,l} - u_{j,l+1} - u_{j,l-1} \end{aligned} \tag{19.5.34}$$

More complicated operators may be similarly split, but there is some art involved. A bad choice of splitting can lead to an algorithm that fails to converge. Usually one tries to base the splitting on the physical nature of the problem. We know for our model problem that an initial transient diffuses away, and we set up the x and y splitting to mimic diffusion in each dimension.

Having chosen a splitting, we difference the time-dependent equation (19.5.31) implicitly in two half-steps:

$$\begin{aligned} \frac{u^{n+1/2} - u^n}{\Delta t/2} &= -\frac{\mathcal{L}_x u^{n+1/2} + \mathcal{L}_y u^n}{\Delta^2} - \rho \\ \frac{u^{n+1} - u^{n+1/2}}{\Delta t/2} &= -\frac{\mathcal{L}_x u^{n+1/2} + \mathcal{L}_y u^{n+1}}{\Delta^2} - \rho \end{aligned} \tag{19.5.35}$$

(cf. equation 19.3.16). Here we have suppressed the spatial indices (j, l). In matrix notation, equations (19.5.35) are

$$(\mathbf{L}_x + r\mathbf{1}) \cdot \mathbf{u}^{n+1/2} = (r\mathbf{1} - \mathbf{L}_y) \cdot \mathbf{u}^n - \Delta^2 \rho \tag{19.5.36}$$

$$(\mathbf{L}_y + r\mathbf{1}) \cdot \mathbf{u}^{n+1} = (r\mathbf{1} - \mathbf{L}_x) \cdot \mathbf{u}^{n+1/2} - \Delta^2 \rho \tag{19.5.37}$$

where

$$r \equiv \frac{2\Delta^2}{\Delta t} \tag{19.5.38}$$

The matrices on the left-hand sides of equations (19.5.36) and (19.5.37) are tridiagonal (and usually positive definite), so the equations can be solved by the

standard tridiagonal algorithm. Given $\mathbf{u}^n$, one solves (19.5.36) for $\mathbf{u}^{n+1/2}$, substitutes on the right-hand side of (19.5.37), and then solves for $\mathbf{u}^{n+1}$. The key question is how to choose the iteration parameter r, the analog of a choice of timestep for an initial value problem.

As usual, the goal is to minimize the spectral radius of the iteration matrix. Although it is beyond our scope to go into details here, it turns out that, for the optimal choice of r, the ADI method has the same rate of convergence as SOR. The individual iteration steps in the ADI method are much more complicated than in SOR, so the ADI method would appear to be inferior. This is in fact true if we choose the same parameter r for every iteration step. However, it is possible to choose a *different* r for each step. If this is done optimally, then ADI is generally more efficient than SOR. We refer you to the literature [1-4] for details.

Our reason for not fully implementing ADI here is that, in most applications, it has been superseded by the multigrid methods described in the next section. Our advice is to use SOR for trivial problems (e.g., 20×20), or for solving a larger problem once only, where ease of programming outweighs expense of computer time. Occasionally, the sparse matrix methods of §2.7 are useful for solving a set of difference equations directly. For production solution of large elliptic problems, however, multigrid is now almost always the method of choice.

CITED REFERENCES AND FURTHER READING:

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6.

Young, D.M. 1971, *Iterative Solution of Large Linear Systems* (New York: Academic Press). [1]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§8.3–8.6. [2]

Varga, R.S. 1962, *Matrix Iterative Analysis* (Englewood Cliffs, NJ: Prentice-Hall). [3]

Spanier, J. 1967, in *Mathematical Methods for Digital Computers, Volume 2* (New York: Wiley), Chapter 11. [4]

19.6 Multigrid Methods for Boundary Value Problems

Practical multigrid methods were first introduced in the 1970s by Brandt. These methods can solve elliptic PDEs discretized on N grid points in $O(N)$ operations. The "rapid" direct elliptic solvers discussed in §19.4 solve special kinds of elliptic equations in $O(N \log N)$ operations. The numerical coefficients in these estimates are such that multigrid methods are comparable to the rapid methods in execution speed. Unlike the rapid methods, however, the multigrid methods can solve general elliptic equations with nonconstant coefficients with hardly any loss in efficiency. Even nonlinear equations can be solved with comparable speed.

Unfortunately there is not a single multigrid algorithm that solves all elliptic problems. Rather there is a multigrid technique that provides the framework for solving these problems. You have to adjust the various components of the algorithm within this framework to solve your specific problem. We can only give a brief

standard tridiagonal algorithm. Given $\mathbf{u}^n$, one solves (19.5.36) for $\mathbf{u}^{n+1/2}$, substitutes on the right-hand side of (19.5.37), and then solves for $\mathbf{u}^{n+1}$. The key question is how to choose the iteration parameter r, the analog of a choice of timestep for an initial value problem.

As usual, the goal is to minimize the spectral radius of the iteration matrix. Although it is beyond our scope to go into details here, it turns out that, for the optimal choice of r, the ADI method has the same rate of convergence as SOR. The individual iteration steps in the ADI method are much more complicated than in SOR, so the ADI method would appear to be inferior. This is in fact true if we choose the same parameter r for every iteration step. However, it is possible to choose a *different* r for each step. If this is done optimally, then ADI is generally more efficient than SOR. We refer you to the literature [1-4] for details.

Our reason for not fully implementing ADI here is that, in most applications, it has been superseded by the multigrid methods described in the next section. Our advice is to use SOR for trivial problems (e.g., 20×20), or for solving a larger problem once only, where ease of programming outweighs expense of computer time. Occasionally, the sparse matrix methods of §2.7 are useful for solving a set of difference equations directly. For production solution of large elliptic problems, however, multigrid is now almost always the method of choice.

CITED REFERENCES AND FURTHER READING:

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6.

Young, D.M. 1971, *Iterative Solution of Large Linear Systems* (New York: Academic Press). [1]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§8.3–8.6. [2]

Varga, R.S. 1962, *Matrix Iterative Analysis* (Englewood Cliffs, NJ: Prentice-Hall). [3]

Spanier, J. 1967, in *Mathematical Methods for Digital Computers, Volume 2* (New York: Wiley), Chapter 11. [4]

19.6 Multigrid Methods for Boundary Value Problems

Practical multigrid methods were first introduced in the 1970s by Brandt. These methods can solve elliptic PDEs discretized on N grid points in $O(N)$ operations. The "rapid" direct elliptic solvers discussed in §19.4 solve special kinds of elliptic equations in $O(N \log N)$ operations. The numerical coefficients in these estimates are such that multigrid methods are comparable to the rapid methods in execution speed. Unlike the rapid methods, however, the multigrid methods can solve general elliptic equations with nonconstant coefficients with hardly any loss in efficiency. Even nonlinear equations can be solved with comparable speed.

Unfortunately there is not a single multigrid algorithm that solves all elliptic problems. Rather there is a multigrid technique that provides the framework for solving these problems. You have to adjust the various components of the algorithm within this framework to solve your specific problem. We can only give a brief

introduction to the subject here. In particular, we will give two sample multigrid routines, one linear and one nonlinear. By following these prototypes and by perusing the references [1-4], you should be able to develop routines to solve your own problems.

There are two related, but distinct, approaches to the use of multigrid techniques. The first, termed "the multigrid method," is a means for speeding up the convergence of a traditional relaxation method, as defined by you on a grid of pre-specified fineness. In this case, you need define your problem (e.g., evaluate its source terms) only on this grid. Other, coarser, grids defined by the method can be viewed as temporary computational adjuncts.

The second approach, termed (perhaps confusingly) "the full multigrid (FMG) method," requires you to be able to define your problem on grids of various sizes (generally by discretizing the same underlying PDE into different-sized sets of finite-difference equations). In this approach, the method obtains successive solutions on finer and finer grids. You can stop the solution either at a pre-specified fineness, or you can monitor the truncation error due to the discretization, quitting only when it is tolerably small.

In this section we will first discuss the "multigrid method," then use the concepts developed to introduce the FMG method. The latter algorithm is the one that we implement in the accompanying programs.

From One-Grid, through Two-Grid, to Multigrid

The key idea of the multigrid method can be understood by considering the simplest case of a two-grid method. Suppose we are trying to solve the linear elliptic problem

$$\mathcal{L}u = f \tag{19.6.1}$$

where $\mathcal{L}$ is some linear elliptic operator and f is the source term. Discretize equation (19.6.1) on a uniform grid with mesh size h. Write the resulting set of linear algebraic equations as

$$\mathcal{L}_h u_h = f_h \tag{19.6.2}$$

Let $\widetilde{u}_h$ denote some approximate solution to equation (19.6.2). We will use the symbol u_h to denote the exact solution to the difference equations (19.6.2). Then the *error* in $\widetilde{u}_h$ or the *correction* is

$$v_h = u_h - \widetilde{u}_h \tag{19.6.3}$$

The *residual* or *defect* is

$$d_h = \mathcal{L}_h \widetilde{u}_h - f_h \tag{19.6.4}$$

(Beware: some authors define residual as minus the defect, and there is not universal agreement about which of these two quantities 19.6.4 defines.) Since $\mathcal{L}_h$ is linear, the error satisfies

$$\mathcal{L}_h v_h = -d_h \tag{19.6.5}$$

At this point we need to make an approximation to $\mathcal{L}_h$ in order to find v_h. The classical iteration methods, such as Jacobi or Gauss-Seidel, do this by finding, at each stage, an approximate solution of the equation

$$\widehat{\mathcal{L}}_h \widehat{v}_h = -d_h \tag{19.6.6}$$

where $\widehat{\mathcal{L}}_h$ is a "simpler" operator than $\mathcal{L}_h$. For example, $\widehat{\mathcal{L}}_h$ is the diagonal part of $\mathcal{L}_h$ for Jacobi iteration, or the lower triangle for Gauss-Seidel iteration. The next approximation is generated by

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \widehat{v}_h \tag{19.6.7}$$

Now consider, as an alternative, a completely different type of approximation for $\mathcal{L}_h$, one in which we "coarsify" rather than "simplify." That is, we form some appropriate approximation $\mathcal{L}_H$ of $\mathcal{L}_h$ on a coarser grid with mesh size H (we will always take $H = 2h$, but other choices are possible). The residual equation (19.6.5) is now approximated by

$$\mathcal{L}_H v_H = -d_H \tag{19.6.8}$$

Since $\mathcal{L}_H$ has smaller dimension, this equation will be easier to solve than equation (19.6.5). To define the defect d_H on the coarse grid, we need a *restriction operator* $\mathcal{R}$ that restricts d_h to the coarse grid:

$$d_H = \mathcal{R} d_h \tag{19.6.9}$$

The restriction operator is also called the *fine-to-coarse operator* or the *injection operator*. Once we have a solution $\widetilde{v}_H$ to equation (19.6.8), we need a *prolongation operator* $\mathcal{P}$ that prolongates or interpolates the correction to the fine grid:

$$\widetilde{v}_h = \mathcal{P} \widetilde{v}_H \tag{19.6.10}$$

The prolongation operator is also called the *coarse-to-fine operator* or the *interpolation operator*. Both $\mathcal{R}$ and $\mathcal{P}$ are chosen to be linear operators. Finally the approximation $\widetilde{u}_h$ can be updated:

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \widetilde{v}_h \tag{19.6.11}$$

One step of this *coarse-grid correction scheme* is thus:

Coarse-Grid Correction

- Compute the defect on the fine grid from (19.6.4).
- Restrict the defect by (19.6.9).
- Solve (19.6.8) exactly on the coarse grid for the correction.
- Interpolate the correction to the fine grid by (19.6.10).

- Compute the next approximation by (19.6.11).

Let's contrast the advantages and disadvantages of relaxation and the coarse-grid correction scheme. Consider the error v_h expanded into a discrete Fourier series. Call the components in the lower half of the frequency spectrum the *smooth components* and the high-frequency components the *nonsmooth components*. We have seen that relaxation becomes very slowly convergent in the limit $h \to 0$, i.e., when there are a large number of mesh points. The reason turns out to be that the smooth components are only slightly reduced in amplitude on each iteration. However, many relaxation methods reduce the amplitude of the nonsmooth components by large factors on each iteration: They are good *smoothing operators*.

For the two-grid iteration, on the other hand, components of the error with wavelengths $\lesssim 2H$ are not even representable on the coarse grid and so cannot be reduced to zero on this grid. But it is exactly these high-frequency components that can be reduced by relaxation on the fine grid! This leads us to combine the ideas of relaxation and coarse-grid correction:

Two-Grid Iteration

- Pre-smoothing: Compute $\bar{u}_h$ by applying $\nu_1 \geq 0$ steps of a relaxation method to $\widetilde{u}_h$.
- Coarse-grid correction: As above, using $\bar{u}_h$ to give $\bar{u}_h^{\rm new}$.
- Post-smoothing: Compute $\widetilde{u}_h^{\rm new}$ by applying $\nu_2 \geq 0$ steps of the relaxation method to $\bar{u}_h^{\rm new}$.

It is only a short step from the above two-grid method to a multigrid method. Instead of solving the coarse-grid defect equation (19.6.8) exactly, we can get an approximate solution of it by introducing an even coarser grid and using the two-grid iteration method. If the convergence factor of the two-grid method is small enough, we will need only a few steps of this iteration to get a good enough approximate solution. We denote the number of such iterations by γ. Obviously we can apply this idea recursively down to some coarsest grid. There the solution is found easily, for example by direct matrix inversion or by iterating the relaxation scheme to convergence.

One iteration of a multigrid method, from finest grid to coarser grids and back to finest grid again, is called a *cycle*. The exact structure of a cycle depends on the value of γ, the number of two-grid iterations at each intermediate stage. The case $\gamma = 1$ is called a V-cycle, while $\gamma = 2$ is called a W-cycle (see Figure 19.6.1). These are the most important cases in practice.

Note that once more than two grids are involved, the pre-smoothing steps after the first one on the finest grid need an initial approximation for the error v. This should be taken to be zero.

Smoothing, Restriction, and Prolongation Operators

The most popular smoothing method, and the one you should try first, is Gauss-Seidel, since it usually leads to a good convergence rate. If we order the mesh points from 1 to N, then the Gauss-Seidel scheme is

$$u_i = -\Big(\sum_{\substack{j=1\\ j\neq i}}^{N} L_{ij}u_j - f_i\Big)\frac{1}{L_{ii}} \qquad i = 1, \ldots, N \tag{19.6.12}$$

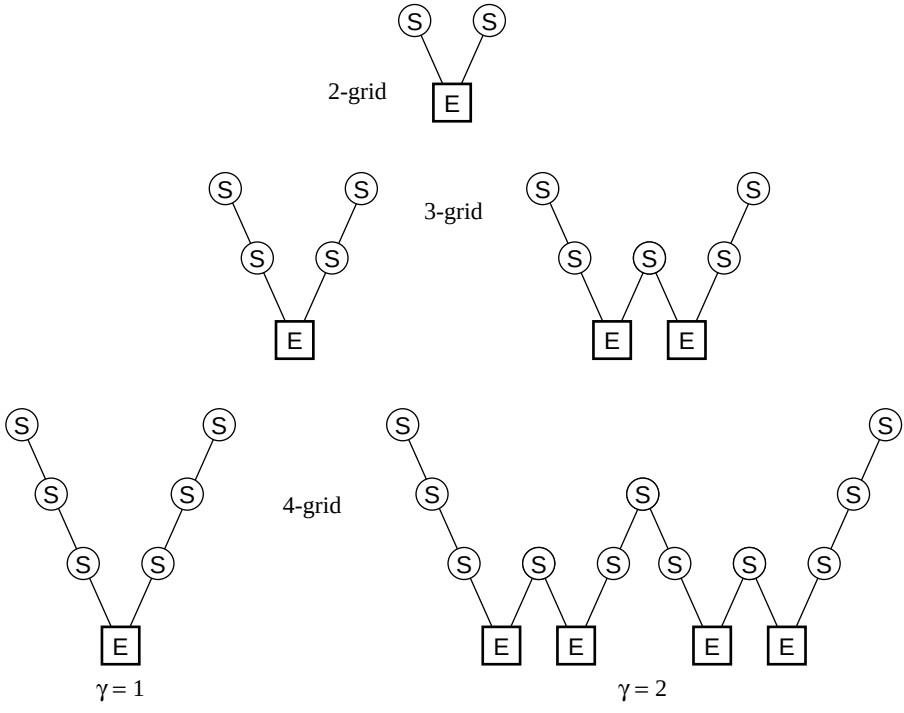


Figure 19.6.1. Structure of multigrid cycles. S denotes smoothing, while E denotes exact solution on the coarsest grid. Each descending line \ denotes restriction ($\mathcal{R}$) and each ascending line / denotes prolongation ($\mathcal{P}$). The finest grid is at the top level of each diagram. For the V-cycles ($\gamma = 1$) the E step is replaced by one 2-grid iteration each time the number of grid levels is increased by one. For the W-cycles ($\gamma = 2$), each E step gets replaced by two 2-grid iterations.

where new values of u are used on the right-hand side as they become available. The exact form of the Gauss-Seidel method depends on the ordering chosen for the mesh points. For typical second-order elliptic equations like our model problem equation (19.0.3), as differenced in equation (19.0.8), it is usually best to use red-black ordering, making one pass through the mesh updating the "even" points (like the red squares of a checkerboard) and another pass updating the "odd" points (the black squares). When quantities are more strongly coupled along one dimension than another, one should relax a whole line along that dimension simultaneously. Line relaxation for nearest-neighbor coupling involves solving a tridiagonal system, and so is still efficient. Relaxing odd and even lines on successive passes is called zebra relaxation and is usually preferred over simple line relaxation.

Note that SOR should *not* be used as a smoothing operator. The overrelaxation destroys the high-frequency smoothing that is so crucial for the multigrid method.

A succint notation for the prolongation and restriction operators is to give their *symbol*. The symbol of $\mathcal{P}$ is found by considering v_H to be 1 at some mesh point (x, y), zero elsewhere, and then asking for the values of $\mathcal{P}v_H$. The most popular prolongation operator is simple bilinear interpolation. It gives nonzero values at the 9 points (x, y), $(x + h, y), \ldots, (x - h, y - h)$, where the values are $1, \frac{1}{2}, \ldots, \frac{1}{4}$.

Its symbol is therefore

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{19.6.13}$$

The symbol of $\mathcal{R}$ is defined by considering v_h to be defined everywhere on the fine grid, and then asking what is $\mathcal{R}v_h$ at (x, y) as a linear combination of these values. The simplest possible choice for $\mathcal{R}$ is *straight injection*, which means simply filling each coarse-grid point with the value from the corresponding fine-grid point. Its symbol is "[1]." However, difficulties can arise in practice with this choice. It turns out that a safe choice for $\mathcal{R}$ is to make it the adjoint operator to $\mathcal{P}$. To define the adjoint, define the scalar product of two grid functions u_h and v_h for mesh size h as

$$\langle u_h | v_h \rangle_h \equiv h^2 \sum_{x,y} u_h(x, y) v_h(x, y) \tag{19.6.14}$$

Then the adjoint of $\mathcal{P}$, denoted $\mathcal{P}^\dagger$, is defined by

$$\langle u_H | \mathcal{P}^\dagger v_h \rangle_H = \langle \mathcal{P} u_H | v_h \rangle_h \tag{19.6.15}$$

Now take $\mathcal{P}$ to be bilinear interpolation, and choose $u_H = 1$ at (x, y), zero elsewhere. Set $\mathcal{P}^\dagger = \mathcal{R}$ in (19.6.15) and $H = 2h$. You will find that

$$(\mathcal{R}v_h)_{(x,y)} = \tfrac{1}{4} v_h(x, y) + \tfrac{1}{8} v_h(x + h, y) + \tfrac{1}{16} v_h(x + h, y + h) + \cdots \tag{19.6.16}$$

so that the symbol of $\mathcal{R}$ is

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{19.6.17}$$

Note the simple rule: The symbol of $\mathcal{R}$ is $\frac{1}{4}$ the transpose of the matrix defining the symbol of $\mathcal{P}$, equation (19.6.13). This rule is general whenever $\mathcal{R} = \mathcal{P}^\dagger$ and $H = 2h$.

The particular choice of $\mathcal{R}$ in (19.6.17) is called *full weighting*. Another popular choice for $\mathcal{R}$ is *half weighting*, "halfway" between full weighting and straight injection. Its symbol is

$$\begin{bmatrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{1}{2} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{bmatrix} \tag{19.6.18}$$

A similar notation can be used to describe the difference operator $\mathcal{L}_h$. For example, the standard differencing of the model problem, equation (19.0.6), is represented by the *five-point difference star*

$$\mathcal{L}_h = \frac{1}{h^2} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{19.6.19}$$

If you are confronted with a new problem and you are not sure what $\mathcal{P}$ and $\mathcal{R}$ choices are likely to work well, here is a safe rule: Suppose m_p is the order of the interpolation $\mathcal{P}$ (i.e., it interpolates polynomials of degree $m_p - 1$ exactly). Suppose m_r is the order of $\mathcal{R}$, and that $\mathcal{R}$ is the adjoint of some $\mathcal{P}$ (not necessarily the $\mathcal{P}$ you intend to use). Then if m is the order of the differential operator $\mathcal{L}_h$, you should satisfy the inequality $m_p + m_r > m$. For example, bilinear interpolation and its adjoint, full weighting, for Poisson's equation satisfy $m_p + m_r = 4 > m = 2$.

Of course the $\mathcal{P}$ and $\mathcal{R}$ operators should enforce the boundary conditions for your problem. The easiest way to do this is to rewrite the difference equation to have homogeneous boundary conditions by modifying the source term if necessary (cf. §19.4). Enforcing homogeneous boundary conditions simply requires the $\mathcal{P}$ operator to produce zeros at the appropriate boundary points. The corresponding $\mathcal{R}$ is then found by $\mathcal{R} = \mathcal{P}^\dagger$.

Full Multigrid Algorithm

So far we have described multigrid as an iterative scheme, where one starts with some initial guess on the finest grid and carries out enough cycles (V-cycles, W-cycles,...) to achieve convergence. This is the simplest way to use multigrid: Simply apply enough cycles until some appropriate convergence criterion is met. However, efficiency can be improved by using the *Full Multigrid Algorithm* (FMG), also known as *nested iteration*.

Instead of starting with an arbitrary approximation on the finest grid (e.g., $u_h = 0$), the first approximation is obtained by interpolating from a coarse-grid solution:

$$u_h = \mathcal{P} u_H \tag{19.6.20}$$

The coarse-grid solution itself is found by a similar FMG process from even coarser grids. At the coarsest level, you start with the exact solution. Rather than proceed as in Figure 19.6.1, then, FMG gets to its solution by a series of increasingly tall "N's," each taller one probing a finer grid (see Figure 19.6.2).

Note that $\mathcal{P}$ in (19.6.20) need not be the same $\mathcal{P}$ used in the multigrid cycles. It should be at least of the same order as the discretization $\mathcal{L}_h$, but sometimes a higher-order operator leads to greater efficiency.

It turns out that you usually need one or at most two multigrid cycles at each level before proceeding down to the next finer grid. While there is theoretical guidance on the required number of cycles (e.g., [2]), you can easily determine it empirically. Fix the finest level and study the solution values as you increase the number of cycles per level. The asymptotic value of the solution is the exact solution of the difference equations. The difference between this exact solution and the solution for a small number of cycles is the iteration error. Now fix the number of cycles to be large, and vary the number of levels, i.e., the smallest value of h used. In this way you can estimate the truncation error for a given h. In your final production code, there is no point in using more cycles than you need to get the iteration error down to the size of the truncation error.

The simple multigrid iteration (cycle) needs the right-hand side f only at the finest level. FMG needs f at all levels. If the boundary conditions are homogeneous,

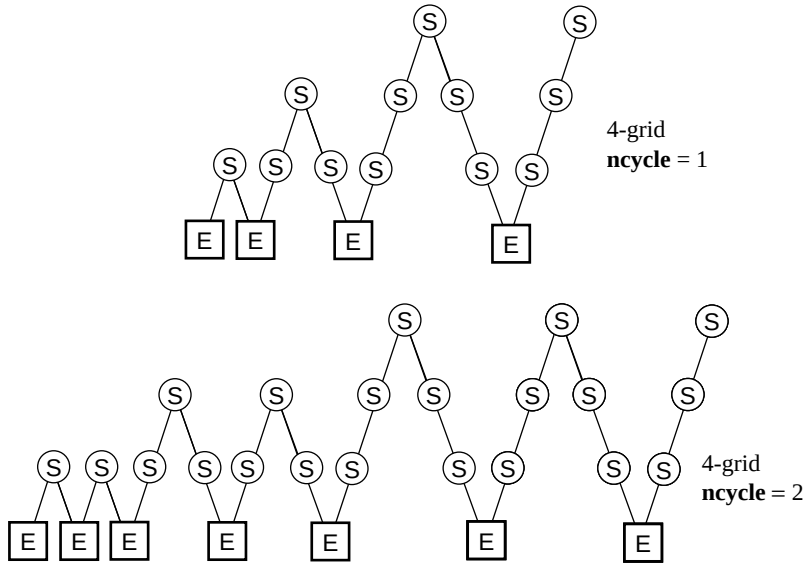


Figure 19.6.2. Structure of cycles for the full multigrid (FMG) method. This method starts on the coarsest grid, interpolates, and then refines (by "V's"), the solution onto grids of increasing fineness.

you can use $f_H = \mathcal{R} f_h$. This prescription is not always safe for inhomogeneous boundary conditions. In that case it is better to discretize f on each coarse grid.

Note that the FMG algorithm produces the solution on all levels. It can therefore be combined with techniques like Richardson extrapolation.

We now give a routine `mglin` that implements the Full Multigrid Algorithm for a linear equation, the model problem (19.0.6). It uses red-black Gauss-Seidel as the smoothing operator, bilinear interpolation for $\mathcal{P}$, and half-weighting for $\mathcal{R}$. To change the routine to handle another linear problem, all you need do is modify the subroutines `relax`, `resid`, and `slvsml` appropriately. A feature of the routine is the dynamical allocation of storage for variables defined on the various grids. The subroutine `maloc` emulates the `C` function `malloc`. It allows you to write subroutines that operate on two-dimensional arrays in the usual way, but to allocate storage for these arrays in the calling program "on the fly" out of a single long one-dimensional array.

```
      SUBROUTINE mglin(u,n,ncycle)
      INTEGER n,ncycle,NPRE,NPOST,NG,MEMLEN
      DOUBLE PRECISION u(n,n)
      PARAMETER (NG=5,MEMLEN=13*2**(2*NG)/3+14*2**NG+8*NG-100/3)
      PARAMETER (NPRE=1,NPOST=1)
C     USES addint,copy,fill0,interp,maloc,relax,resid,rstrct,slvsml
```

Full Multigrid Algorithm for solution of linear elliptic equation, here the model problem (19.0.6). On input `u(1:n,1:n)` contains the right-hand side ρ, while on output it returns the solution. The dimension `n` is related to the number of grid levels used in the solution, `NG` below, by `n` $= 2$`**NG` $+ 1$. `ncycle` is the number of V-cycles to be used at each level. Parameters: `NG` is the number of grid levels used; `MEMLEN` is the maximum amount of memory that can be allocated by calls to `maloc`; `NPRE` and `NPOST` are the number of relaxation sweeps before and after the coarse-grid correction is computed.

```
      INTEGER j,jcycle,jj,jpost,jpre,mem,nf,ngrid,nn,ires(NG),
     *     irho(NG),irhs(NG),iu(NG),maloc
      DOUBLE PRECISION z
```

```
      COMMON /memory/ z(MEMLEN),mem                Storage for grid functions is allocated by maloc
      mem=0                                            from array z.
      nn=n/2+1
      ngrid=NG-1
      irho(ngrid)=maloc(nn**2)                     Allocate storage for r.h.s. on grid NG − 1,
      call rstrct(z(irho(ngrid)),u,nn)             and fill it by restricting from the fine grid.
1     if (nn.gt.3) then                            Similarly allocate storage and fill r.h.s. on all
          nn=nn/2+1                                    coarse grids.
          ngrid=ngrid-1
          irho(ngrid)=maloc(nn**2)
          call rstrct(z(irho(ngrid)),z(irho(ngrid+1)),nn)
      goto 1
      endif
      nn=3
      iu(1)=maloc(nn**2)
      irhs(1)=maloc(nn**2)
      call slvsml(z(iu(1)),z(irho(1)))             Initial solution on coarsest grid.
      ngrid=NG
      do 16 j=2,ngrid                              Nested iteration loop.
          nn=2*nn-1
          iu(j)=maloc(nn**2)
          irhs(j)=maloc(nn**2)
          ires(j)=maloc(nn**2)
          call interp(z(iu(j)),z(iu(j-1)),nn)      Interpolate from coarse grid to next finer grid.
          if (j.ne.ngrid) then
              call copy(z(irhs(j)),z(irho(j)),nn)  Set up r.h.s.
          else
              call copy(z(irhs(j)),u,nn)
          endif
          do 15 jcycle=1,ncycle                    V-cycle loop.
           nf=nn
           do 12 jj=j,2,-1                         Downward stoke of the V.
              do 11 jpre=1,NPRE                    Pre-smoothing.
               call relax(z(iu(jj)),z(irhs(jj)),nf)
              enddo 11
              call resid(z(ires(jj)),z(iu(jj)),z(irhs(jj)),nf)
              nf=nf/2+1
              call rstrct(z(irhs(jj-1)),z(ires(jj)),nf)
                Restriction of the residual is the next r.h.s.
              call fill0(z(iu(jj-1)),nf)           Zero for initial guess in next relaxation.
           enddo 12
           call slvsml(z(iu(1)),z(irhs(1)))        Bottom of V: solve on coarsest grid.
           nf=3
           do 14 jj=2,j                            Upward stroke of V.
              nf=2*nf-1
              call addint(z(iu(jj)),z(iu(jj-1)),z(ires(jj)),nf)
                Use res for temporary storage inside addint.
              do 13 jpost=1,NPOST                  Post-smoothing.
               call relax(z(iu(jj)),z(irhs(jj)),nf)
              enddo 13
           enddo 14
          enddo 15
      enddo 16
      call copy(u,z(iu(ngrid)),n)                  Return solution in u.
      return
      END


      SUBROUTINE rstrct(uc,uf,nc)
      INTEGER nc
      DOUBLE PRECISION uc(nc,nc),uf(2*nc-1,2*nc-1)
          Half-weighting restriction. nc is the coarse-grid dimension. The fine-grid solution is input
          in uf(1:2*nc-1,1:2*nc-1), the coarse-grid solution is returned in uc(1:nc,1:nc).
      INTEGER ic,if,jc,jf
```

```
      do 12 jc=2,nc-1                                   Interior points.
        jf=2*jc-1
        do 11 ic=2,nc-1
          if=2*ic-1
          uc(ic,jc)=.5d0*uf(if,jf)+.125d0*(uf(if+1,jf)+
     *         uf(if-1,jf)+uf(if,jf+1)+uf(if,jf-1))
        enddo 11
      enddo 12
      do 13 ic=1,nc                                     Boundary points.
        uc(ic,1)=uf(2*ic-1,1)
        uc(ic,nc)=uf(2*ic-1,2*nc-1)
      enddo 13
      do 14 jc=1,nc
        uc(1,jc)=uf(1,2*jc-1)
        uc(nc,jc)=uf(2*nc-1,2*jc-1)
      enddo 14
      return
      END


      SUBROUTINE interp(uf,uc,nf)
      INTEGER nf
      DOUBLE PRECISION uc(nf/2+1,nf/2+1),uf(nf,nf)
      INTEGER ic,if,jc,jf,nc
        Coarse-to-fine prolongation by bilinear interpolation. nf is the fine-grid dimension. The
        coarse-grid solution is input as uc(1:nc,1:nc), where nc = nf/2 + 1. The fine-grid
        solution is returned in uf(1:nf,1:nf).
      nc=nf/2+1
      do 12 jc=1,nc                                     Do elements that are copies.
        jf=2*jc-1
        do 11 ic=1,nc
          uf(2*ic-1,jf)=uc(ic,jc)
        enddo 11
      enddo 12
      do 14 jf=1,nf,2                                   Do odd-numbered columns, interpolating ver-
        do 13 if=2,nf-1,2                                 tically.
          uf(if,jf)=.5d0*(uf(if+1,jf)+uf(if-1,jf))
        enddo 13
      enddo 14
      do 16 jf=2,nf-1,2                                 Do even-numbered columns, interpolating hor-
        do 15 if=1,nf                                     izontally.
          uf(if,jf)=.5d0*(uf(if,jf+1)+uf(if,jf-1))
        enddo 15
      enddo 16
      return
      END


      SUBROUTINE addint(uf,uc,res,nf)
      INTEGER nf
      DOUBLE PRECISION res(nf,nf),uc(nf/2+1,nf/2+1),uf(nf,nf)
C     USES interp
        Does coarse-to-fine interpolation and adds result to uf. nf is the fine-grid dimension. The
        coarse-grid solution is input as uc(1:nc,1:nc), where nc = nf/2 + 1. The fine-grid
        solution is returned in uf(1:nf,1:nf). res(1:nf,1:nf) is used for temporary storage.
      INTEGER i,j
      call interp(res,uc,nf)
      do 12 j=1,nf
        do 11 i=1,nf
          uf(i,j)=uf(i,j)+res(i,j)
        enddo 11
      enddo 12
      return
      END
```

```
      SUBROUTINE slvsml(u,rhs)
      DOUBLE PRECISION rhs(3,3),u(3,3)
C     USES fill0
```

Solution of the model problem on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in `rhs(1:3,1:3)` and the solution is returned in `u(1:3,1:3)`.

```
      DOUBLE PRECISION h
      call fill0(u,3)
      h=.5d0
      u(2,2)=-h*h*rhs(2,2)/4.d0
      return
      END


      SUBROUTINE relax(u,rhs,n)
      INTEGER n
      DOUBLE PRECISION rhs(n,n),u(n,n)
```

Red-black Gauss-Seidel relaxation for model problem. The current value of the solution `u(1:n,1:n)` is updated, using the right-hand side function `rhs(1:n,1:n)`.

```
      INTEGER i,ipass,isw,j,jsw
      DOUBLE PRECISION h,h2
      h=1.d0/(n-1)
      h2=h*h
      jsw=1
      do 13 ipass=1,2                          Red and black sweeps.
        isw=jsw
        do 12 j=2,n-1
          do 11 i=isw+1,n-1,2                  Gauss-Seidel formula.
            u(i,j)=0.25d0*(u(i+1,j)+u(i-1,j)+u(i,j+1)
     *           +u(i,j-1)-h2*rhs(i,j))
          enddo 11
          isw=3-isw
        enddo 12
        jsw=3-jsw
      enddo 13
      return
      END


      SUBROUTINE resid(res,u,rhs,n)
      INTEGER n
      DOUBLE PRECISION res(n,n),rhs(n,n),u(n,n)
```

Returns *minus* the residual for the model problem. Input quantities are `u(1:n,1:n)` and `rhs(1:n,1:n)`, while `res(1:n,1:n)` is returned.

```
      INTEGER i,j
      DOUBLE PRECISION h,h2i
      h=1.d0/(n-1)
      h2i=1.d0/(h*h)
      do 12 j=2,n-1                            Interior points.
        do 11 i=2,n-1
          res(i,j)=-h2i*(u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j-1)-
     *         4.d0*u(i,j))+rhs(i,j)
        enddo 11
      enddo 12
      do 13 i=1,n                              Boundary points.
        res(i,1)=0.d0
        res(i,n)=0.d0
        res(1,i)=0.d0
        res(n,i)=0.d0
      enddo 13
      return
      END
```

```
      SUBROUTINE copy(aout,ain,n)
      INTEGER n
      DOUBLE PRECISION ain(n,n),aout(n,n)
         Copies ain(1:n,1:n) to aout(1:n,1:n).
      INTEGER i,j
      do 12 i=1,n
          do 11 j=1,n
              aout(j,i)=ain(j,i)
          enddo 11
      enddo 12
      return
      END



      SUBROUTINE fill0(u,n)
      INTEGER n
      DOUBLE PRECISION u(n,n)
         Fills u(1:n,1:n) with zeros.
      INTEGER i,j
      do 12 j=1,n
          do 11 i=1,n
              u(i,j)=0.d0
          enddo 11
      enddo 12
      return
      END



      FUNCTION maloc(len)
      INTEGER maloc,len,NG,MEMLEN
      PARAMETER (NG=5,MEMLEN=13*2**(2*NG)/3+14*2**NG+8*NG-100/3)     for mglin
C     PARAMETER (NG=5,MEMLEN=17*2**(2*NG)/3+18*2**NG+10*NG-86/3)     for mgfas, N.B.!
      INTEGER mem
      DOUBLE PRECISION z
      COMMON /memory/ z(MEMLEN),mem
         Dynamical storage allocation. Returns integer pointer to the starting position for len array
         elements in the array z. The preceding array element is filled with the value of len, and
         the variable mem is updated to point to the last element of z that has been used.
      if (mem+len+1.gt.MEMLEN) pause 'insufficient memory in maloc'
      z(mem+1)=len
      maloc=mem+2
      mem=mem+len+1
      return
      END
```

The routine `mglin` is written for clarity, not maximum efficiency, so that it is easy to modify. Several simple changes will speed up the execution time:

- The defect d_h vanishes identically at all black mesh points after a red-black Gauss-Seidel step. Thus $d_H = \mathcal{R}d_h$ for half-weighting reduces to simply copying half the defect from the fine grid to the corresponding coarse-grid point. The calls to `resid` followed by `rstrct` in the first part of the V-cycle can be replaced by a routine that loops only over the coarse grid, filling it with half the defect.
- Similarly, the quantity $\widetilde{u}_h^{\rm new} = \widetilde{u}_h + \mathcal{P}\widetilde{v}_H$ need not be computed at red mesh points, since they will immediately be redefined in the subsequent Gauss-Seidel sweep. This means that `addint` need only loop over black points.

- You can speed up `relax` in several ways. First, you can have a special form when the initial guess is zero, and omit the routine `fill0`. Next, you can store $h^2 f_h$ on the various grids and save a multiplication. Finally, it is possible to save an addition in the Gauss-Seidel formula by rewriting it with intermediate variables.
- On typical problems, `mglin` with `ncycle` $= 1$ will return a solution with the iteration error bigger than the truncation error for the given size of h. To knock the error down to the size of the truncation error, you have to set `ncycle` $= 2$ or, more cheaply, `npre` $= 2$. A more efficient way turns out to be to use a higher-order $\mathcal{P}$ in (19.6.20) than the linear interpolation used in the V-cycle.

Implementing all the above features typically gives up to a factor of two improvement in execution time and is certainly worthwhile in a production code.

Nonlinear Multigrid: The FAS Algorithm

Now turn to solving a nonlinear elliptic equation, which we write symbolically as

$$\mathcal{L}(u) = 0 \tag{19.6.21}$$

Any explicit source term has been moved to the left-hand side. Suppose equation (19.6.21) is suitably discretized:

$$\mathcal{L}_h(u_h) = 0 \tag{19.6.22}$$

We will see below that in the multigrid algorithm we will have to consider equations where a nonzero right-hand side is generated during the course of the solution:

$$\mathcal{L}_h(u_h) = f_h \tag{19.6.23}$$

One way of solving nonlinear problems with multigrid is to use Newton's method, which produces linear equations for the correction term at each iteration. We can then use linear multigrid to solve these equations. A great strength of the multigrid idea, however, is that it can be applied *directly* to nonlinear problems. All we need is a suitable *nonlinear* relaxation method to smooth the errors, plus a procedure for approximating corrections on coarser grids. This direct approach is Brandt's Full Approximation Storage Algorithm (FAS). No nonlinear equations need be solved, except perhaps on the coarsest grid.

To develop the nonlinear algorithm, suppose we have a relaxation procedure that can smooth the residual vector as we did in the linear case. Then we can seek a smooth correction v_h to solve (19.6.23):

$$\mathcal{L}_h(\tilde{u}_h + v_h) = f_h \tag{19.6.24}$$

To find v_h, note that

$$\begin{aligned} \mathcal{L}_h(\tilde{u}_h + v_h) - \mathcal{L}_h(\tilde{u}_h) &= f_h - \mathcal{L}_h(\tilde{u}_h) \\ &= -d_h \end{aligned} \tag{19.6.25}$$

The right-hand side is smooth after a few nonlinear relaxation sweeps. Thus we can transfer the left-hand side to a coarse grid:

$$\mathcal{L}_H(u_H) - \mathcal{L}_H(\mathcal{R}\tilde{u}_h) = -\mathcal{R}d_h \tag{19.6.26}$$

that is, we solve

$$\mathcal{L}_H(u_H) = \mathcal{L}_H(\mathcal{R}\tilde{u}_h) - \mathcal{R}d_h \tag{19.6.27}$$

on the coarse grid. (This is how nonzero right-hand sides appear.) Suppose the approximate solution is $\tilde{u}_H$. Then the coarse-grid correction is

$$\tilde{v}_H = \tilde{u}_H - \mathcal{R}\tilde{u}_h \tag{19.6.28}$$

and

$$\widetilde{u}_h^{\rm new} = \widetilde{u}_h + \mathcal{P}(\widetilde{u}_H - \mathcal{R}\widetilde{u}_h) \tag{19.6.29}$$

Note that $\mathcal{P}\mathcal{R} \neq 1$ in general, so $\widetilde{u}_h^{\rm new} \neq \mathcal{P}\widetilde{u}_H$. This is a key point: In equation (19.6.29) the interpolation error comes only from the correction, not from the full solution $\widetilde{u}_H$.

Equation (19.6.27) shows that one is solving for the full approximation u_H, not just the error as in the linear algorithm. This is the origin of the name FAS.

The FAS multigrid algorithm thus looks very similar to the linear multigrid algorithm. The only differences are that both the defect d_h and the relaxed approximation u_h have to be restricted to the coarse grid, where now it is equation (19.6.27) that is solved by recursive invocation of the algorithm. However, instead of implementing the algorithm this way, we will first describe the so-called *dual viewpoint*, which leads to a powerful alternative way of looking at the multigrid idea.

The dual viewpoint considers the *local truncation error*, defined as

$$\tau \equiv \mathcal{L}_h(u) - f_h \tag{19.6.30}$$

where u is the exact solution of the oiginal continuum equation. If we rewrite this as

$$\mathcal{L}_h(u) = f_h + \tau \tag{19.6.31}$$

we see that τ can be regarded as the correction to f_h so that the solution of the fine-grid equation will be the exact solution u.

Now consider the *relative truncation error* τ_h, which is defined on the H-grid relative to the h-grid:

$$\tau_h \equiv \mathcal{L}_H(\mathcal{R}u_h) - \mathcal{R}\mathcal{L}_h(u_h) \tag{19.6.32}$$

Since $\mathcal{L}_h(u_h) = f_h$, this can be rewritten as

$$\mathcal{L}_H(u_H) = f_H + \tau_h \tag{19.6.33}$$

In other words, we can think of τ_h as the correction to f_H that makes the solution of the coarse-grid equation equal to the fine-grid solution. Of course we cannot compute τ_h, but we do have an approximation to it from using $\widetilde{u}_h$ in equation (19.6.32):

$$\tau_h \simeq \widetilde{\tau}_h \equiv \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}\mathcal{L}_h(\widetilde{u}_h) \tag{19.6.34}$$

Replacing τ_h by $\widetilde{\tau}_h$ in equation (19.6.33) gives

$$\mathcal{L}_H(u_H) = \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}d_h \tag{19.6.35}$$

which is just the coarse-grid equation (19.6.27)!

Thus we see that there are two complementary viewpoints for the relation between coarse and fine grids:

- Coarse grids are used to accelerate the convergence of the smooth components of the fine-grid residuals.
- Fine grids are used to compute correction terms to the coarse-grid equations, yielding fine-grid accuracy on the coarse grids.

One benefit of this new viewpoint is that it allows us to derive a natural stopping criterion for a multigrid iteration. Normally the criterion would be

$$\|d_h\| \leq \epsilon \tag{19.6.36}$$

and the question is how to choose ϵ. There is clearly no benefit in iterating beyond the point when the remaining error is dominated by the local truncation error τ. The computable quantity is $\widetilde{\tau}_h$. What is the relation between τ and $\widetilde{\tau}_h$? For the typical case of a second-order accurate differencing scheme,

$$\tau = \mathcal{L}_h(u) - \mathcal{L}_h(u_h) = h^2\tau_2(x,y) + \cdots \tag{19.6.37}$$

Assume the solution satisfies $u_h = u + h^2 u_2(x,y) + \cdots$. Then, assuming $\mathcal{R}$ is of high enough order that we can neglect its effect, equation (19.6.32) gives

$$\begin{aligned}\tau_h &\simeq \mathcal{L}_H(u + h^2 u_2) - \mathcal{L}_h(u + h^2 u_2) \\ &= \mathcal{L}_H(u) - \mathcal{L}_h(u) + h^2[\mathcal{L}'_H(u_2) - \mathcal{L}'_h(u_2)] + \cdots \\ &= (H^2 - h^2)\tau_2 + O(h^4)\end{aligned} \tag{19.6.38}$$

For the usual case of $H = 2h$ we therefore have

$$\tau \simeq \tfrac{1}{3}\tau_h \simeq \tfrac{1}{3}\widetilde{\tau}_h \tag{19.6.39}$$

The stopping criterion is thus equation (19.6.36) with

$$\epsilon = \alpha\|\widetilde{\tau}_h\|, \qquad \alpha \sim \tfrac{1}{3} \tag{19.6.40}$$

We have one remaining task before implementing our nonlinear multigrid algorithm: choosing a nonlinear relaxation scheme. Once again, your first choice should probably be the nonlinear Gauss-Seidel scheme. If the discretized equation (19.6.23) is written with some choice of ordering as

$$L_i(u_1, \ldots, u_N) = f_i, \qquad i = 1, \ldots, N \tag{19.6.41}$$

then the nonlinear Gauss-Seidel schemes solves

$$L_i(u_1, \ldots, u_{i-1}, u_i^{\rm new}, u_{i+1}, \ldots, u_N) = f_i \tag{19.6.42}$$

for $u_i^{\rm new}$. As usual new u's replace old u's as soon as they have been computed. Often equation (19.6.42) is linear in $u_i^{\rm new}$, since the nonlinear terms are discretized by means of its neighbors. If this is not the case, we replace equation (19.6.42) by one step of a Newton iteration:

$$u_i^{\rm new} = u_i^{\rm old} - \frac{L_i(u_i^{\rm old}) - f_i}{\partial L_i(u_i^{\rm old})/\partial u_i} \tag{19.6.43}$$

For example, consider the simple nonlinear equation

$$\nabla^2 u + u^2 = \rho \tag{19.6.44}$$

In two-dimensional notation, we have

$$\mathcal{L}(u_{i,j}) = (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j})/h^2 + u_{i,j}^2 - \rho_{i,j} = 0 \tag{19.6.45}$$

Since

$$\frac{\partial \mathcal{L}}{\partial u_{i,j}} = -4/h^2 + 2u_{i,j} \tag{19.6.46}$$

the Newton Gauss-Seidel iteration is

$$u_{i,j}^{\rm new} = u_{i,j} - \frac{\mathcal{L}(u_{i,j})}{-4/h^2 + 2u_{i,j}} \tag{19.6.47}$$

Here is a routine `mgfas` that solves equation (19.6.44) using the Full Multigrid Algorithm and the FAS scheme. Restriction and prolongation are done as in `mglin`. We have included the convergence test based on equation (19.6.40). A successful multigrid solution of a problem should aim to satisfy this condition with the maximum number of V-cycles, `maxcyc`, equal to 1 or 2. The routine `mgfas` uses the same subroutines `copy`, `interp`, `maloc`, and `rstrct` as `mglin`, but with a larger storage requirement `MEMLEN` in `maloc` (be sure to change the `PARAMETER` statement in that routine, as indicated by the commented line).

```
      SUBROUTINE mgfas(u,n,maxcyc)
      INTEGER maxcyc,n,NPRE,NPOST,NG,MEMLEN
      DOUBLE PRECISION u(n,n),ALPHA
      PARAMETER (NG=5,MEMLEN=17*2**(2*NG)/3+18*2**NG+10*NG-86/3)
      PARAMETER (NPRE=1,NPOST=1,ALPHA=.33d0)
C     USES anorm2,copy,interp,lop,maloc,matadd,matsub,relax2,rstrct,slvsm2
```

Full Multigrid Algorithm for FAS solution of nonlinear elliptic equation, here equation (19.6.44). On input `u(1:n,1:n)` contains the right-hand side ρ, while on output it returns the solution. The dimension `n` is related to the number of grid levels used in the solution, `NG` below, by `n` $= 2^{**}$`NG` $+ 1$. `maxcyc` is the maximum number of V-cycles to be used at each level.

Parameters: `NG` is the number of grid levels used; `MEMLEN` is the maximum amount of memory that can be allocated by calls to `maloc`; `NPRE` and `NPOST` are the number of relaxation sweeps before and after the coarse-grid correction is computed; `ALPHA` relates the estimated truncation error to the norm of the residual.

```
      INTEGER j,jcycle,jj,jm1,jpost,jpre,mem,nf,ngrid,nn,irho(NG),
     *     irhs(NG),itau(NG),itemp(NG),iu(NG),maloc
      DOUBLE PRECISION res,trerr,z,anorm2
      COMMON /memory/ z(MEMLEN),mem                 Storage for grid functions is allocated by maloc
      mem=0                                           from array z.
      nn=n/2+1
      ngrid=NG-1
      irho(ngrid)=maloc(nn**2)                      Allocate storage for r.h.s. on grid NG − 1,
      call rstrct(z(irho(ngrid)),u,nn)              and fill it by restricting from the fine grid.
1     if (nn.gt.3) then                             Similarly allocate storage and fill r.h.s. on all
        nn=nn/2+1                                     coarse grids.
        ngrid=ngrid-1
        irho(ngrid)=maloc(nn**2)
        call rstrct(z(irho(ngrid)),z(irho(ngrid+1)),nn)
      goto 1
      endif
      nn=3
      iu(1)=maloc(nn**2)
      irhs(1)=maloc(nn**2)
      itau(1)=maloc(nn**2)
      itemp(1)=maloc(nn**2)
      call slvsm2(z(iu(1)),z(irho(1)))              Initial solution on coarsest grid.
      ngrid=NG
      do 16 j=2,ngrid                               Nested iteration loop.
        nn=2*nn-1
        iu(j)=maloc(nn**2)
        irhs(j)=maloc(nn**2)
        itau(j)=maloc(nn**2)
        itemp(j)=maloc(nn**2)
        call interp(z(iu(j)),z(iu(j-1)),nn)         Interpolate from coarse grid to next finer grid.
        if (j.ne.ngrid) then
          call copy(z(irhs(j)),z(irho(j)),nn)       Set up r.h.s.
        else
          call copy(z(irhs(j)),u,nn)
        endif
        do 15 jcycle=1,maxcyc                       V-cycle loop.
         nf=nn
         do 12 jj=j,2,-1                            Downward stoke of the V.
           do 11 jpre=1,NPRE                        Pre-smoothing.
            call relax2(z(iu(jj)),z(irhs(jj)),nf)
           enddo 11
           call lop(z(itemp(jj)),z(iu(jj)),nf)                  L_h(ũ_h).
           nf=nf/2+1
           jm1=jj-1
           call rstrct(z(itemp(jm1)),z(itemp(jj)),nf)           RL_h(ũ_h).
           call rstrct(z(iu(jm1)),z(iu(jj)),nf)                 Rũ_h.
           call lop(z(itau(jm1)),z(iu(jm1)),nf)     L_H(Rũ_h) stored temporarily in τ̃_h.
           call matsub(z(itau(jm1)),z(itemp(jm1)),z(itau(jm1)),nf)   Form τ̃_h.
           if(jj.eq.j)trerr=ALPHA*anorm2(z(itau(jm1)),nf)   Estimate truncation error τ.
```

```
        call rstrct(z(irhs(jm1)),z(irhs(jj)),nf)          f_H.
        call matadd(z(irhs(jm1)),z(itau(jm1)),z(irhs(jm1)),nf)   f_H + τ̃_h.
      enddo 12
      call slvsm2(z(iu(1)),z(irhs(1)))      Bottom of V: Solve on coarsest grid.
      nf=3
      do 14 jj=2,j                          Upward stroke of V.
        jm1=jj-1
        call rstrct(z(itemp(jm1)),z(iu(jj)),nf)          Rũ_h.
        call matsub(z(iu(jm1)),z(itemp(jm1)),z(itemp(jm1)),nf)   ũ_H − Rũ_h.
        nf=2*nf-1
        call interp(z(itau(jj)),z(itemp(jm1)),nf)        P(ũ_H − Rũ_h) stored in τ̃_h.
        call matadd(z(iu(jj)),z(itau(jj)),z(iu(jj)),nf)  Form ũ_h^new.
        do 13 jpost=1,NPOST                 Post-smoothing.
         call relax2(z(iu(jj)),z(irhs(jj)),nf)
        enddo 13
      enddo 14
      call lop(z(itemp(j)),z(iu(j)),nf)     Form residual ‖d_h‖.
      call matsub(z(itemp(j)),z(irhs(j)),z(itemp(j)),nf)
      res=anorm2(z(itemp(j)),nf)
      if(res.lt.trerr)goto 2                No more V-cycles needed if residual small
    enddo 15                                    enough.
2   continue
  enddo 16
  call copy(u,z(iu(ngrid)),n)               Return solution in u.
  return
  END


  SUBROUTINE relax2(u,rhs,n)
  INTEGER n
  DOUBLE PRECISION rhs(n,n),u(n,n)
    Red-black Gauss-Seidel relaxation for equation (19.6.44). The current value of the solution
    u(1:n,1:n) is updated, using the right-hand side function rhs(1:n,1:n).
  INTEGER i,ipass,isw,j,jsw
  DOUBLE PRECISION foh2,h,h2i,res
  h=1.d0/(n-1)
  h2i=1.d0/(h*h)
  foh2=-4.d0*h2i
  jsw=1
  do 13 ipass=1,2                                   Red and black sweeps.
    isw=jsw
    do 12 j=2,n-1
      do 11 i=isw+1,n-1,2
        res=h2i*(u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j-1)-
*            4.d0*u(i,j))+u(i,j)**2-rhs(i,j)
        u(i,j)=u(i,j)-res/(foh2+2.d0*u(i,j))  Newton Gauss-Seidel formula.
      enddo 11
      isw=3-isw
    enddo 12
    jsw=3-jsw
  enddo 13
  return
  END


  SUBROUTINE slvsm2(u,rhs)
  DOUBLE PRECISION rhs(3,3),u(3,3)
C USES fill0
    Solution of equation (19.6.44) on the coarsest grid, where h = 1/2. The right-hand side is
    input in rhs(1:3,1:3) and the solution is returned in u(1:3,1:3).
  DOUBLE PRECISION disc,fact,h
  call fill0(u,3)
  h=.5d0
  fact=2.d0/h**2
  disc=sqrt(fact**2+rhs(2,2))
```

```
      u(2,2)=-rhs(2,2)/(fact+disc)
      return
      END


      SUBROUTINE lop(out,u,n)
      INTEGER n
      DOUBLE PRECISION out(n,n),u(n,n)
        Given u(1:n,1:n), returns L_h(ũ_h) for equation (19.6.44) in out(1:n,1:n).
      INTEGER i,j
      DOUBLE PRECISION h,h2i
      h=1.d0/(n-1)
      h2i=1.d0/(h*h)
      do 12 j=2,n-1                                   Interior points.
        do 11 i=2,n-1
          out(i,j)=h2i*(u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j-1)-
     *         4.d0*u(i,j))+u(i,j)**2
        enddo 11
      enddo 12
      do 13 i=1,n                                     Boundary points.
        out(i,1)=0.d0
        out(i,n)=0.d0
        out(1,i)=0.d0
        out(n,i)=0.d0
      enddo 13
      return
      END


      SUBROUTINE matadd(a,b,c,n)
      INTEGER n
      DOUBLE PRECISION a(n,n),b(n,n),c(n,n)
        Adds a(1:n,1:n) to b(1:n,1:n) and returns result in c(1:n,1:n).
      INTEGER i,j
      do 12 j=1,n
        do 11 i=1,n
          c(i,j)=a(i,j)+b(i,j)
        enddo 11
      enddo 12
      return
      END


      SUBROUTINE matsub(a,b,c,n)
      INTEGER n
      DOUBLE PRECISION a(n,n),b(n,n),c(n,n)
        Subtracts b(1:n,1:n) from a(1:n,1:n) and returns result in c(1:n,1:n).
      INTEGER i,j
      do 12 j=1,n
        do 11 i=1,n
          c(i,j)=a(i,j)-b(i,j)
        enddo 11
      enddo 12
      return
      END


      DOUBLE PRECISION FUNCTION anorm2(a,n)
      INTEGER n
      DOUBLE PRECISION a(n,n)
        Returns the Euclidean norm of the matrix a(1:n,1:n).
      INTEGER i,j
      DOUBLE PRECISION sum
      sum=0.d0
      do 12 j=1,n
        do 11 i=1,n
```

```
        sum=sum+a(i,j)**2
    enddo 11
enddo 12
anorm2=sqrt(sum)/n
return
END
```

CITED REFERENCES AND FURTHER READING:

Brandt, A. 1977, *Mathematics of Computation*, vol. 31, pp. 333–390. [1]

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag). [2]

Stuben, K., and Trottenberg, U. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag), pp. 1–176. [3]

Brandt, A. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag). [4]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill).

Briggs, W.L. 1987, *A Multigrid Tutorial* (Philadelphia: S.I.A.M.).

Jespersen, D. 1984, *Multrigrid Methods for Partial Differential Equations* (Washington: Mathematical Association of America).

McCormick, S.F. (ed.) 1988, *Multigrid Methods: Theory, Applications, and Supercomputing* (New York: Marcel Dekker).

Hackbusch, W., and Trottenberg, U. (eds.) 1991, *Multigrid Methods III* (Boston: Birkhauser).

Wesseling, P. 1992, *An Introduction to Multigrid Methods* (New York: Wiley).

Chapter 20. Less-Numerical Algorithms

20.0 Introduction

You can stop reading now. You are done with *Numerical Recipes*, as such. This final chapter is an idiosyncratic collection of "*less*-numerical recipes" which, for one reason or another, we have decided to include between the covers of an otherwise *more*-numerically oriented book. Authors of computer science texts, we've noticed, like to throw in a token numerical subject (usually quite a dull one — quadrature, for example). We find that we are not free of the reverse tendency.

Our selection of material is not completely arbitrary. One topic, Gray codes, was already used in the construction of quasi-random sequences (§7.7), and here needs only some additional explication. Two other topics, on diagnosing a computer's floating-point parameters, and on arbitrary precision arithmetic, give additional insight into the machinery behind the casual assumption that computers are useful for doing things with numbers (as opposed to bits or characters). The latter of these topics also shows a very different use for Chapter 12's fast Fourier transform.

The three other topics (checksums, Huffman and arithmetic coding) involve different aspects of data coding, compression, and validation. If you handle a large amount of data — numerical data, even — then a passing familiarity with these subjects might at some point come in handy. In §13.6, for example, we already encountered a good use for Huffman coding.

But again, you don't have to read this chapter. (And you should learn about quadrature from Chapters 4 and 16, not from a computer science text!)

20.1 Diagnosing Machine Parameters

A convenient fiction is that a computer's floating-point arithmetic is "accurate enough." If you believe this fiction, then numerical analysis becomes a very clean subject. Roundoff error disappears from view; many finite algorithms become "exact"; only docile truncation error (§1.2) stands between you and a perfect calculation. Sounds rather naive, doesn't it?

Yes, it is naive. Notwithstanding, it is a fiction necessarily adopted throughout most of this book. To do a good job of answering the question of how roundoff error

Chapter 20. Less-Numerical Algorithms

20.0 Introduction

You can stop reading now. You are done with *Numerical Recipes*, as such. This final chapter is an idiosyncratic collection of "*less*-numerical recipes" which, for one reason or another, we have decided to include between the covers of an otherwise *more*-numerically oriented book. Authors of computer science texts, we've noticed, like to throw in a token numerical subject (usually quite a dull one — quadrature, for example). We find that we are not free of the reverse tendency.

Our selection of material is not completely arbitrary. One topic, Gray codes, was already used in the construction of quasi-random sequences (§7.7), and here needs only some additional explication. Two other topics, on diagnosing a computer's floating-point parameters, and on arbitrary precision arithmetic, give additional insight into the machinery behind the casual assumption that computers are useful for doing things with numbers (as opposed to bits or characters). The latter of these topics also shows a very different use for Chapter 12's fast Fourier transform.

The three other topics (checksums, Huffman and arithmetic coding) involve different aspects of data coding, compression, and validation. If you handle a large amount of data — numerical data, even — then a passing familiarity with these subjects might at some point come in handy. In §13.6, for example, we already encountered a good use for Huffman coding.

But again, you don't have to read this chapter. (And you should learn about quadrature from Chapters 4 and 16, not from a computer science text!)

20.1 Diagnosing Machine Parameters

A convenient fiction is that a computer's floating-point arithmetic is "accurate enough." If you believe this fiction, then numerical analysis becomes a very clean subject. Roundoff error disappears from view; many finite algorithms become "exact"; only docile truncation error (§1.2) stands between you and a perfect calculation. Sounds rather naive, doesn't it?

Yes, it is naive. Notwithstanding, it is a fiction necessarily adopted throughout most of this book. To do a good job of answering the question of how roundoff error

propagates, or can be bounded, for every algorithm that we have discussed would be impractical. In fact, it would not be possible: Rigorous analysis of many practical algorithms has never been made, by us or anyone.

Proper numerical analysts cringe when they hear a user say, "I was getting roundoff errors with single precision, so I switched to double." The actual meaning is, "for this particular algorithm, and my particular data, double precision *seemed* able to restore my erroneous belief in the 'convenient fiction'." We admit that most of the mentions of precision or roundoff in *Numerical Recipes* are only slightly more quantitative in character. That comes along with our trying to be "practical."

It is important to know what the limitations of your machine's floating-point arithmetic actually are — the more so when your treatment of floating-point roundoff error is going to be intuitive, experimental, or casual. Methods for determining useful floating-point parameters experimentally have been developed by Cody [1], Malcolm [2], and others, and are embodied in the routine `machar`, below, which follows Cody's implementation.

All of `machar`'s arguments are returned values. Here is what they mean:

- `ibeta` (called B in §1.2) is the radix in which numbers are represented, almost always 2, but occasionally 16, or even 10.
- `it` is the number of base-`ibeta` digits in the floating-point mantissa M (see Figure 1.2.1).
- `machep` is the exponent of the smallest (most negative) power of `ibeta` that, added to 1.0, gives something different from 1.0.
- `eps` is the floating-point number $\texttt{ibeta}^{\texttt{machep}}$, loosely referred to as the "floating-point precision."
- `negep` is the exponent of the smallest power of `ibeta` that, subtracted from 1.0, gives something different from 1.0.
- `epsneg` is $\texttt{ibeta}^{\texttt{negep}}$, another way of defining floating-point precision. Not infrequently `epsneg` is 0.5 times `eps`; occasionally `eps` and `epsneg` are equal.
- `iexp` is the number of bits in the exponent (including its sign or bias).
- `minexp` is the smallest (most negative) power of `ibeta` consistent with there being no leading zeros in the mantissa.
- `xmin` is the floating-point number $\texttt{ibeta}^{\texttt{minexp}}$, generally the smallest (in magnitude) useable floating value.
- `maxexp` is the smallest (positive) power of `ibeta` that causes overflow.
- `xmax` is $(1-\texttt{epsneg}) \times \texttt{ibeta}^{\texttt{maxexp}}$, generally the largest (in magnitude) useable floating value.
- `irnd` returns a code in the range $0 \ldots 5$, giving information on what kind of rounding is done in addition, and on how underflow is handled. See below.
- `ngrd` is the number of "guard digits" used when truncating the product of two mantissas to fit the representation.

There is a lot of subtlety in a program like `machar`, whose purpose is to ferret out machine properties that are supposed to be transparent to the user. Further, it must do so avoiding error conditions, like overflow and underflow, that might interrupt its execution. In some cases the program is able to do this only by recognizing certain characteristics of "standard" representations. For example, it recognizes the IEEE standard representation [3] by its rounding behavior, and assumes certain features of its exponent representation as a consequence. We refer you to [1] and

Sample Results Returned by `machar`			
	typical IEEE-compliant machine		DEC VAX
precision	single	double	single
`ibeta`	2	2	2
`it`	24	53	24
`machep`	-23	-52	-24
`eps`	1.19×10^{-7}	2.22×10^{-16}	5.96×10^{-8}
`negep`	-24	-53	-24
`epsneg`	5.96×10^{-8}	1.11×10^{-16}	5.96×10^{-8}
`iexp`	8	11	8
`minexp`	-126	-1022	-128
`xmin`	1.18×10^{-38}	2.23×10^{-308}	2.94×10^{-39}
`maxexp`	128	1024	127
`xmax`	3.40×10^{38}	1.79×10^{308}	1.70×10^{38}
`irnd`	5	5	1
`ngrd`	0	0	0

references therein for details. Be aware that `machar` can give incorrect results on some nonstandard machines.

The parameter `irnd` needs some additional explanation. In the IEEE standard, bit patterns correspond to exact, "representable" numbers. The specified method for rounding an addition is to add two representable numbers "exactly," and then round the sum to the closest representable number. If the sum is precisely halfway between two representable numbers, it should be rounded to the even one (low-order bit zero). The same behavior should hold for all the other arithmetic operations, that is, they should be done in a manner equivalent to infinite precision, and then rounded to the closest representable number.

If `irnd` returns 2 or 5, then your computer is compliant with this standard. If it returns 1 or 4, then it is doing some kind of rounding, but not the IEEE standard. If `irnd` returns 0 or 3, then it is truncating the result, not rounding it — not desirable.

The other issue addressed by `irnd` concerns underflow. If a floating value is less than `xmin`, many computers underflow its value to zero. Values `irnd` $= 0, 1,$ or 2 indicate this behavior. The IEEE standard specifies a more graceful kind of underflow: As a value becomes smaller than `xmin`, its exponent is frozen at the smallest allowed value, while its mantissa is decreased, acquiring leading zeros and "gracefully" losing precision. This is indicated by `irnd` $= 3, 4,$ or 5.

```
      SUBROUTINE machar(ibeta,it,irnd,ngrd,machep,negep,iexp,minexp,
     *     maxexp,eps,epsneg,xmin,xmax)
      INTEGER ibeta,iexp,irnd,it,machep,maxexp,minexp,negep,ngrd
      REAL eps,epsneg,xmax,xmin
```

Determines and returns machine-specific parameters affecting floating-point arithmetic. Returned values include `ibeta`, the floating-point radix; `it`, the number of base-`ibeta` digits in the floating-point mantissa; `eps`, the smallest positive number that, added to 1.0, is not equal to 1.0; `epsneg`, the smallest positive number that, subtracted from 1.0, is not equal to 1.0; `xmin`, the smallest representable positive number; and `xmax`, the largest representable positive number. See text for description of other returned parameters.

```
      INTEGER i,itemp,iz,j,k,mx,nxres
      REAL a,b,beta,betah,betain,one,t,temp,temp1,tempa,two,y,z
     *     ,zero,CONV
      CONV(i)=float(i)              Change to dble(i), and change REAL declaration above to
      one=CONV(1)                       DOUBLE PRECISION to find double precision parameters.
      two=one+one
      zero=one-one
      a=one                         Determine ibeta and beta by the method of M. Malcolm.
1     continue
          a=a+a
          temp=a+one
          temp1=temp-a
      if (temp1-one.eq.zero) goto 1
      b=one
2     continue
          b=b+b
          temp=a+b
          itemp=int(temp-a)
      if (itemp.eq.0) goto 2
      ibeta=itemp
      beta=CONV(ibeta)
      it=0                          Determine it and irnd.
      b=one
3     continue
          it=it+1
          b=b*beta
          temp=b+one
          temp1=temp-b
      if (temp1-one.eq.zero) goto 3
      irnd=0
      betah=beta/two
      temp=a+betah
      if (temp-a.ne.zero) irnd=1
      tempa=a+beta
      temp=tempa+betah
      if ((irnd.eq.0).and.(temp-tempa.ne.zero)) irnd=2
      negep=it+3                    Determine negep and epsneg.
      betain=one/beta
      a=one
      do 11 i=1, negep
          a=a*betain
      enddo 11
      b=a
4     continue
          temp=one-a
          if (temp-one.ne.zero) goto 5
          a=a*beta
          negep=negep-1
      goto 4
5     negep=-negep
      epsneg=a
      machep=-it-3                  Determine machep and eps.
      a=b
6     continue
```

```
      temp=one+a
      if (temp-one.ne.zero) goto 7
      a=a*beta
      machep=machep+1
    goto 6
7   eps=a
    ngrd=0                          Determine ngrd.
    temp=one+eps
    if ((irnd.eq.0).and.(temp*one-one.ne.zero)) ngrd=1
    i=0                             Determine iexp.
    k=1
    z=betain
    t=one+eps
    nxres=0
8   continue                        Loop until an underflow occurs, then exit.
      y=z
      z=y*y
      a=z*one                       Check here for the underflow.
      temp=z*t
      if ((a+a.eq.zero).or.(abs(z).ge.y)) goto 9
      temp1=temp*betain
      if (temp1*beta.eq.z) goto 9
      i=i+1
      k=k+k
    goto 8
9   if (ibeta.ne.10) then
      iexp=i+1
      mx=k+k
    else                            For decimal machines only.
      iexp=2
      iz=ibeta
10    if (k.ge.iz) then
        iz=iz*ibeta
        iexp=iexp+1
      goto 10
      endif
      mx=iz+iz-1
    endif
20  xmin=y                          To determine minexp and xmin, loop until an underflow oc-
    y=y*betain                          curs, then exit.
    a=y*one                         Check here for the underflow.
    temp=y*t
    if (((a+a).ne.zero).and.(abs(y).lt.xmin)) then
      k=k+1
      temp1=temp*betain
      if ((temp1*beta.ne.y).or.(temp.eq.y)) then
        goto 20
      else
        nxres=3
        xmin=y
      endif
    endif
    minexp=-k                       Determine maxexp, xmax.
    if ((mx.le.k+k-3).and.(ibeta.ne.10)) then
      mx=mx+mx
      iexp=iexp+1
    endif
    maxexp=mx+minexp
    irnd=irnd+nxres                 Adjust irnd to reflect partial underflow.
    if (irnd.ge.2) maxexp=maxexp-2          Adjust for IEEE-style machines.
    i=maxexp+minexp
      Adjust for machines with implicit leading bit in binary mantissa, and machines with radix
      point at extreme right of mantissa.
    if ((ibeta.eq.2).and.(i.eq.0)) maxexp=maxexp-1
```

```
if (i.gt.20) maxexp=maxexp-1
if (a.ne.y) maxexp=maxexp-2
xmax=one-epsneg
if (xmax*one.ne.xmax) xmax=one-beta*epsneg
xmax=xmax/(beta*beta*beta*xmin)
i=maxexp+minexp+3
do 12 j=1,i
     if (ibeta.eq.2) xmax=xmax+xmax
     if (ibeta.ne.2) xmax=xmax*beta
enddo 12
return
END
```

Some typical values returned by `machar` are given in the table, above. IEEE-compliant machines referred to in the table include most UNIX workstations (SUN, DEC, MIPS), and Apple Macintosh IIs. IBM PCs with floating co-processors are generally IEEE-compliant, except that some compilers underflow intermediate results ungracefully, yielding `irnd` $= 2$ rather than 5. Notice, as in the case of a VAX (fourth column), that representations with a "phantom" leading 1 bit in the mantissa achieve a smaller `eps` for the same wordlength, but cannot underflow gracefully.

CITED REFERENCES AND FURTHER READING:

Goldberg, D. 1991, *ACM Computing Surveys*, vol. 23, pp. 5–48.

Cody, W.J. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 303–311. [1]

Malcolm, M.A. 1972, *Communications of the ACM*, vol. 15, pp. 949–951. [2]

IEEE Standard for Binary Floating-Point Numbers, ANSI/IEEE Std 754–1985 (New York: IEEE, 1985). [3]

20.2 Gray Codes

A Gray code is a function $G(i)$ of the integers i, that for each integer $N \geq 0$ is one-to-one for $0 \leq i \leq 2^N - 1$, and that has the following remarkable property: The binary representation of $G(i)$ and $G(i+1)$ differ in *exactly one bit*. An example of a Gray code (in fact, the most commonly used one) is the sequence 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, and 1000, for $i = 0, \ldots, 15$. The algorithm for generating this code is simply to form the bitwise exclusive-or (XOR) of i with $i/2$ (integer part). Think about how the carries work when you add one to a number in binary, and you will be able to see why this works. You will also see that $G(i)$ and $G(i+1)$ differ in the bit position of the rightmost zero bit of i (prefixing a leading zero if necessary).

The spelling is "Gray," not "gray": The codes are named after one Frank Gray, who first patented the idea for use in shaft encoders. A shaft encoder is a wheel with concentric coded stripes each of which is "read" by a fixed conducting brush. The idea is to generate a binary code describing the angle of the wheel. The obvious, but wrong, way to build a shaft encoder is to have one stripe (the innermost, say) conducting on half the wheel, but insulating on the other half; the next stripe is conducting in quadrants 1 and 3; the next stripe is conducting in octants 1, 3, 5, and 7; and so on. The brushes together then read a direct binary code for the position of the wheel.

```
if (i.gt.20) maxexp=maxexp-1
if (a.ne.y) maxexp=maxexp-2
xmax=one-epsneg
if (xmax*one.ne.xmax) xmax=one-beta*epsneg
xmax=xmax/(beta*beta*beta*xmin)
i=maxexp+minexp+3
do 12 j=1,i
     if (ibeta.eq.2) xmax=xmax+xmax
     if (ibeta.ne.2) xmax=xmax*beta
enddo 12
return
END
```

Some typical values returned by `machar` are given in the table, above. IEEE-compliant machines referred to in the table include most UNIX workstations (SUN, DEC, MIPS), and Apple Macintosh IIs. IBM PCs with floating co-processors are generally IEEE-compliant, except that some compilers underflow intermediate results ungracefully, yielding `irnd` $= 2$ rather than 5. Notice, as in the case of a VAX (fourth column), that representations with a "phantom" leading 1 bit in the mantissa achieve a smaller `eps` for the same wordlength, but cannot underflow gracefully.

CITED REFERENCES AND FURTHER READING:

Goldberg, D. 1991, *ACM Computing Surveys*, vol. 23, pp. 5–48.

Cody, W.J. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 303–311. [1]

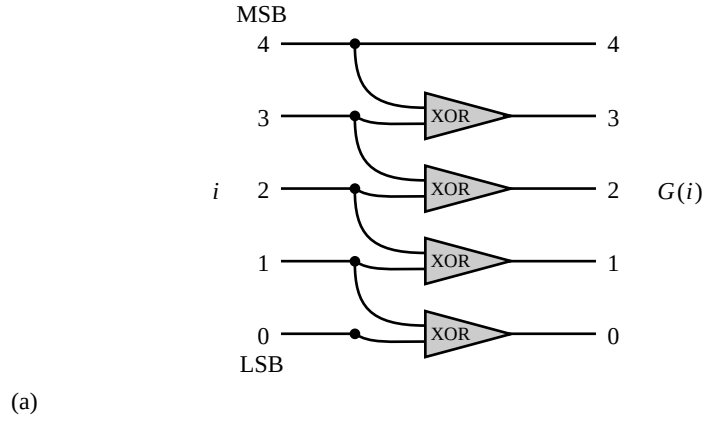
Malcolm, M.A. 1972, *Communications of the ACM*, vol. 15, pp. 949–951. [2]

IEEE Standard for Binary Floating-Point Numbers, ANSI/IEEE Std 754–1985 (New York: IEEE, 1985). [3]

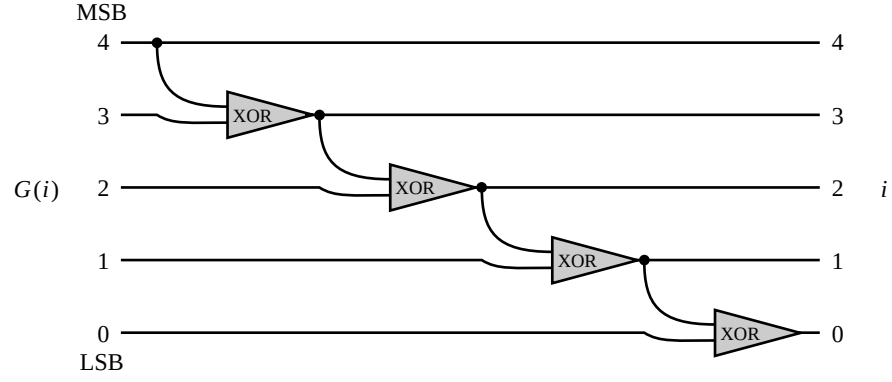
20.2 Gray Codes

A Gray code is a function $G(i)$ of the integers i, that for each integer $N \geq 0$ is one-to-one for $0 \leq i \leq 2^N - 1$, and that has the following remarkable property: The binary representation of $G(i)$ and $G(i+1)$ differ in *exactly one bit*. An example of a Gray code (in fact, the most commonly used one) is the sequence 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, and 1000, for $i = 0, \ldots, 15$. The algorithm for generating this code is simply to form the bitwise exclusive-or (XOR) of i with $i/2$ (integer part). Think about how the carries work when you add one to a number in binary, and you will be able to see why this works. You will also see that $G(i)$ and $G(i+1)$ differ in the bit position of the rightmost zero bit of i (prefixing a leading zero if necessary).

The spelling is "Gray," not "gray": The codes are named after one Frank Gray, who first patented the idea for use in shaft encoders. A shaft encoder is a wheel with concentric coded stripes each of which is "read" by a fixed conducting brush. The idea is to generate a binary code describing the angle of the wheel. The obvious, but wrong, way to build a shaft encoder is to have one stripe (the innermost, say) conducting on half the wheel, but insulating on the other half; the next stripe is conducting in quadrants 1 and 3; the next stripe is conducting in octants 1, 3, 5, and 7; and so on. The brushes together then read a direct binary code for the position of the wheel.



(a)



(b)

Figure 20.2.1. Single-bit operations for calculating the Gray code $G(i)$ from i (a), or the inverse (b). LSB and MSB indicate the least and most significant bits, respectively. XOR denotes exclusive-or.

The reason this method is bad, is that there is no way to guarantee that all the brushes will make or break contact *exactly* simultaneously as the wheel turns. Going from position 7 (0111) to 8 (1000), one might pass spuriously and transiently through 6 (0110), 14 (1110), and 10 (1010), as the different brushes make or break contact. Use of a Gray code on the encoding stripes guarantees that there is no transient state between 7 (0100 in the sequence above) and 8 (1100).

Of course we then need circuitry, or algorithmics, to translate from $G(i)$ to i. Figure 20.2.1 (b) shows how this is done by a cascade of XOR gates. The idea is that each output bit should be the XOR of all more significant input bits. To do N bits of Gray code inversion requires $N-1$ steps (or gate delays) in the circuit. (Nevertheless, this is typically very fast in circuitry.) In a register with word-wide binary operations, we don't have to do N consecutive operations, but only $\ln_2 N$. The trick is to use the associativity of XOR and group the operations hierarchically. This involves sequential right-shifts by $1, 2, 4, 8, \ldots$ bits until the wordlength is exhausted. Here is a piece of code for doing both $G(i)$ and its inverse.

```
      FUNCTION igray(n,is)
      INTEGER igray,is,n
        For zero or positive values of is, return the Gray code of n; if is is negative, return the
        inverse Gray code of n.
      INTEGER idiv,ish
      if (is.ge.0) then              This is the easy direction!
        igray=ieor(n,n/2)
      else                           This is the more complicated direction: In hierarchical stages,
        ish=-1                           starting with a one-bit right shift, cause each bit to be
        igray=n                          XORed with all more significant bits.
1       continue
          idiv=ishft(igray,ish)
          igray=ieor(igray,idiv)
          if(idiv.le.1.or.ish.eq.-16)return
          ish=ish+ish                Double the amount of shift on the next cycle.
        goto 1
      endif
      return
      END
```

In numerical work, Gray codes can be useful when you need to do some task that depends intimately on the bits of i, looping over many values of i. Then, if there are economies in repeating the task for values differing by only one bit, it makes sense to do things in Gray code order rather than consecutive order. We saw an example of this in §7.7, for the generation of quasi-random sequences.

CITED REFERENCES AND FURTHER READING:

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (New York: Cambridge University Press), §8.02.

Knuth, D.E. *Combinatorial Algorithms*, vol. 4 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §7.2.1. [Unpublished. Will it be always so?]

20.3 Cyclic Redundancy and Other Checksums

When you send a sequence of bits from point A to point B, you want to know that it will arrive without error. A common form of insurance is the "parity bit," attached to 7-bit ASCII characters to put them into 8-bit format. The parity bit is chosen so as to make the total number of one-bits (versus zero-bits) either always even ("even parity") or always odd ("odd parity"). Any *single bit* error in a character will thereby be detected. When errors are sufficiently rare, and do not occur closely bunched in time, use of parity provides sufficient error detection.

Unfortunately, in real situations, a single noise "event" is likely to disrupt more than one bit. Since the parity bit has two possible values (0 and 1), it gives, on average, only a 50% chance of detecting an erroneous character with more than one wrong bit. That probability, 50%, is not nearly good enough for most applications. Most communications protocols [1] use a multibit generalization of the parity bit called a "cyclic redundancy check" or CRC. In typical applications the CRC is 16 bits long (two bytes or two characters), so that the chance of a random error going undetected is 1 in $2^{16} = 65536$. Moreover, M-bit CRCs have the mathematical property of detecting *all* errors that occur in M or fewer *consecutive* bits, for any

```
      FUNCTION igray(n,is)
      INTEGER igray,is,n
        For zero or positive values of is, return the Gray code of n; if is is negative, return the
        inverse Gray code of n.
      INTEGER idiv,ish
      if (is.ge.0) then              This is the easy direction!
        igray=ieor(n,n/2)
      else                           This is the more complicated direction: In hierarchical stages,
        ish=-1                           starting with a one-bit right shift, cause each bit to be
        igray=n                          XORed with all more significant bits.
1       continue
          idiv=ishft(igray,ish)
          igray=ieor(igray,idiv)
          if(idiv.le.1.or.ish.eq.-16)return
          ish=ish+ish                Double the amount of shift on the next cycle.
        goto 1
      endif
      return
      END
```

In numerical work, Gray codes can be useful when you need to do some task that depends intimately on the bits of i, looping over many values of i. Then, if there are economies in repeating the task for values differing by only one bit, it makes sense to do things in Gray code order rather than consecutive order. We saw an example of this in §7.7, for the generation of quasi-random sequences.

CITED REFERENCES AND FURTHER READING:

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (New York: Cambridge University Press), §8.02.

Knuth, D.E. *Combinatorial Algorithms*, vol. 4 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §7.2.1. [Unpublished. Will it be always so?]

20.3 Cyclic Redundancy and Other Checksums

When you send a sequence of bits from point A to point B, you want to know that it will arrive without error. A common form of insurance is the "parity bit," attached to 7-bit ASCII characters to put them into 8-bit format. The parity bit is chosen so as to make the total number of one-bits (versus zero-bits) either always even ("even parity") or always odd ("odd parity"). Any *single bit* error in a character will thereby be detected. When errors are sufficiently rare, and do not occur closely bunched in time, use of parity provides sufficient error detection.

Unfortunately, in real situations, a single noise "event" is likely to disrupt more than one bit. Since the parity bit has two possible values (0 and 1), it gives, on average, only a 50% chance of detecting an erroneous character with more than one wrong bit. That probability, 50%, is not nearly good enough for most applications. Most communications protocols [1] use a multibit generalization of the parity bit called a "cyclic redundancy check" or CRC. In typical applications the CRC is 16 bits long (two bytes or two characters), so that the chance of a random error going undetected is 1 in $2^{16} = 65536$. Moreover, M-bit CRCs have the mathematical property of detecting *all* errors that occur in M or fewer *consecutive* bits, for any

length of message. (We prove this below.) Since noise in communication channels tends to be "bursty," with short sequences of adjacent bits getting corrupted, this consecutive-bit property is highly desirable.

Normally CRCs lie in the province of communications software experts and chip-level hardware designers — people with bits under their fingernails. However, there are at least two kinds of situations where some understanding of CRCs can be useful to the rest of us. First, we sometimes need to be able to communicate with a lower-level piece of hardware or software that expects a valid CRC as part of its input. For example, it can be convenient to have a program generate XMODEM or Kermit [2] packets directly into the communications line rather than having to store the data in a local file.

Second, in the manipulation of large quantities of (e.g., experimental) data, it is useful to be able to tag aggregates of data (whether numbers, records, lines, or whole files) with a statistically unique "key," its CRC. Aggregates of any size can then be compared for identity by comparing only their short CRC keys. Differing keys imply nonidentical records. Identical keys imply, to high statistical certainty, identical records. If you can't tolerate the very small probability of being wrong, you can do a full comparison of the records when the keys are identical. When there is a possibility of files or data records being inadvertently or irresponsibly modified (for example, by a computer virus), it is useful to have their prior CRCs stored externally on a physically secure medium, like a floppy disk.

Sometimes CRCs can be used to compress data as it is recorded. If identical data records occur frequently, one can keep sorted in memory the CRCs of previously encountered records. A new record is archived in full if its CRC is different, otherwise only a pointer to a previous record need be archived. In this application one might desire a 4- or 8-byte CRC, to make the odds of mistakenly discarding a different data record be tolerably small; or, if previous records can be randomly accessed, a full comparison can be made to decide whether records with identical CRCs are in fact identical.

Now let us briefly discuss the theory of CRCs. After that, we will give implementations of various (related) CRCs that are used by the official or de facto standard protocols [1-3] listed in the accompanying table.

The mathematics underlying CRCs is "polynomials over the integers modulo 2." Any binary message can be thought of as a polynomial with coefficients 0 and 1. For example, the message "1100001101" is the polynomial $x^9 + x^8 + x^3 + x^2 + 1$. Since 0 and 1 are the only integers modulo 2, a power of x in the polynomial is either present (1) or absent (0). A polynomial over the integers modulo 2 may be irreducible, meaning that it can't be factored. A subset of the irreducible polynomials are the "primitive" polynomials. These generate maximum length sequences when used in shift registers, as described in §7.4. The polynomial $x^2 + 1$ is not irreducible: $x^2+1 = (x+1)(x+1)$, so it is also not primitive. The polynomial $x^4+x^3+x^2+x+1$ is irreducible, but it turns out not to be primitive. The polynomial $x^4 + x + 1$ is both irreducible and primitive.

An M-bit long CRC is based on a particular primitive polynomial of degree M, called the generator polynomial. The choice of which primitive polynomial to use is only a matter of convention. For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the "CCITT polynomial," which is $x^{16} + x^{12} + x^5 + 1$. This polynomial is used by all of the

Conventions and Test Values for Various CRC Protocols

	`icrc` args		Test Values (C_2C_1 *in hex*)		Packet	
Protocol	`jinit`	`jrev`	T	CatMouse987654321	*Format*	*CRC*
XMODEM	0	1	1A71	E556	$S_1S_2 \ldots S_N C_2 C_1$	0
X.25	255	−1	1B26	F56E	$S_1S_2 \ldots S_N \overline{C_1 C_2}$	F0B8
(no name)	255	−1	1B26	F56E	$S_1S_2 \ldots S_N C_1 C_2$	0
SDLC (IBM)	same as X.25					
HDLC (ISO)	same as X.25					
CRC-CCITT	0	−1	14A1	C28D	$S_1S_2 \ldots S_N C_1 C_2$	0
(no name)	0	−1	14A1	C28D	$S_1S_2 \ldots S_N \overline{C_1 C_2}$	F0B8
Kermit	same as CRC-CCITT				see Notes	

Notes: Overbar denotes bit complement. $S_1 \ldots S_N$ are character data. C_1 is CRC's least significant 8 bits, C_2 is its most significant 8 bits, so $CRC = 256\,C_2 + C_1$ (shown in hex). Kermit (block check level 3) sends the CRC as 3 printable ASCII characters (sends value +32). These contain, respectively, 4 most significant bits, 6 middle bits, 6 least significant bits.

protocols listed in the table. Another common choice is the "CRC-16" polynomial $x^{16} + x^{15} + x^2 + 1$, which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, "CRC-12," is $x^{12} + x^{11} + x^3 + x + 1$. A common 32-bit choice, "AUTODIN-II," is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. For a table of some other primitive polynomials, see §7.4.

Given the generator polynomial G of degree M (which can be written either in polynomial form or as a bit-string, e.g., 10001000000100001 for CCITT), here is how you compute the CRC for a sequence of bits S: First, multiply S by x^M, that is, append M zero bits to it. Second divide — by long division — G into Sx^M. Keep in mind that the subtractions in the long division are done modulo 2, so that there are never any "borrows": Modulo 2 subtraction is the same as logical exclusive-or (XOR). Third, ignore the quotient you get. Fourth, when you eventually get to a remainder, it is the CRC, call it C. C will be a polynomial of degree $M - 1$ or less, otherwise you would not have finished the long division. Therefore, in bit string form, it has M bits, which may include leading zeros. (C might even be all zeros, see below.) See [3] for a worked example.

If you work through the above steps in an example, you will see that most of what you write down in the long-division tableau is superfluous. You are actually just left-shifting sequential bits of S, from the right, into an M-bit register. Every time a 1 bit gets shifted off the left end of this register, you zap the register by an XOR with the M low order bits of G (that is, all the bits of G except its leading 1). When a 0 bit is shifted off the left end you don't zap the register. When the last bit that was originally part of S gets shifted off the left end of the register, what remains is the CRC.

You can immediately recognize how efficiently this procedure can be implemented in hardware. It requires only a shift register with a few hard-wired XOR taps into it. That is how CRCs are computed in communications devices, by a single chip (or small part of one). In software, the implementation is not so elegant, since

bit-shifting is not generally very efficient. One therefore typically finds (as in our implementation below) table-driven routines that pre-calculate the result of a bunch of shifts and XORs, say for each of 256 possible 8-bit inputs [4].

We can now see how the CRC gets its ability to detect all errors in M consecutive bits. Suppose two messages, S and T, differ only within a frame of M bits. Then their CRCs differ by an amount that is the remainder when G is divided into $(S-T)x^M \equiv D$. Now D has the form of leading zeros (which can be ignored), followed by some 1's in an M-bit frame, followed by trailing zeros (which are just multiplicative factors of x). Since factorization is unique, G cannot possibly divide D: G is primitive of degree M, while D is a power of x times a factor of (at most) degree $M-1$. Therefore S and T have inevitably different CRCs.

In most protocols, a transmitted block of data consists of some N data bits, directly followed by the M bits of their CRC (or the CRC XORed with a constant, see below). There are two equivalent ways of validating a block at the receiving end. Most obviously, the receiver can compute the CRC of the data bits, and compare it to the transmitted CRC bits. Less obviously, but more elegantly, the receiver can simply compute the CRC of the total block, with $N+M$ bits, and verify that a result of zero is obtained. Proof: The total block is the polynomial $Sx^M + C$ (data left-shifted to make room for the CRC bits). The definition of C is that $Sx^m = QG + C$, where Q is the discarded quotient. But then $Sx^M + C = QG + C + C = QG$ (remember modulo 2), which is a perfect multiple of G. It remains a multiple of G when it gets multiplied by an additional x^M on the receiving end, so it has a zero CRC, q.e.d.

A couple of small variations on the basic procedure need to be mentioned [1,3]: First, when the CRC is computed, the M-bit register need not be initialized to zero. Initializing it to some other M-bit value (e.g., all 1's) in effect prefaces all blocks by a phantom message that would have given the initialization value as its remainder. It is advantageous to do this, since the CRC described thus far otherwise cannot detect the addition or removal of any number of initial zero bits. (Loss of an initial bit, or insertion of zero bits, are common "clocking errors.") Second, one can add (XOR) any M-bit constant K to the CRC before it is transmitted. This constant can either be XORed away at the receiving end, or else it just changes the expected CRC of the whole block by a known amount, namely the remainder of dividing G into Kx^M. The constant K is frequently "all bits," changing the CRC into its ones complement. This has the advantage of detecting another kind of error that the CRC would otherwise not find: deletion of an initial 1 bit in the message with spurious insertion of a 1 bit at the end of the block.

The accompanying function `icrc` implements the above CRC calculation, including the possibility of the mentioned variations. Input to the function is the starting address of an array of characters, and the length of that array. (In practice, `FORTRAN` allows you to use the address of *any* data structure; `icrc` will treat it as a byte array.) Output is in both of two formats. The function value returns the CRC as a 4-byte integer in the range 0 to 65535. The character array `crc`, of length 2, returns the CRC as two 8-bit characters. `icrc` has two "switch" arguments that specify variations in the CRC calculation. A zero or positive value of `jinit` causes the 16-bit register to have each byte initialized with the value `jinit`. A negative value of `jrev` causes each input character to be interpreted as its bit-reverse image, and a similar bit reversal to be done on the output CRC. You do not have to understand this; just use the values of `jinit` and `jrev` specified in the table.

(If you *insist* on knowing, the explanation is that serial data ports send characters *least-significant bit first* (!), and many protocols shift bits into the CRC register in exactly the order received.) The table shows how to construct a block of characters from the input array and output CRC of `icrc`. You should not need to do any additional bit-reversal outside of `icrc`.

The switch `jinit` has one additional use: When negative it causes the input value of the array `crc` to be used as initialization of the register. If `crc` is unmodified since the last call to `icrc`, this in effect appends the current input array to that of the previous call or calls. Use this feature, for example, to build up the CRC of a whole file a line at a time, without keeping the whole file in memory.

At initialization, the routine `icrc` figures out the order in which the bytes occur when a 4-byte character array is equivalenced to a 4-byte integer. This is not strictly portable `FORTRAN`, but it should work on all machines with 32-bit word lengths. `icrc` is loosely based on a more portable C function in [4], a good place to turn if you have trouble running the program here.

Here is how to understand the operation of `icrc`: First look at the function `icrc1`. This incorporates one input character into a 16-bit CRC register. The only trick used is that character bits are XORed into the most significant bits, eight at a time, instead of being fed into the least significant bit, one bit at a time, at the time of the register shift. This works because XOR is associative and commutative — we can feed in character bits *any* time before they will determine whether to zap with the generator polynomial. (The decimal constant 4129 has the generator's bits in it.)

```
FUNCTION icrc1(crc,onech,ib1,ib2,ib3)
INTEGER icrc1,ib1,ib2,ib3
    Given a remainder up to now, return the new CRC after one character is added. This routine
    is functionally equivalent to icrc(,,1,-1,1), but slower. It is used by icrc to initialize
    its table.
INTEGER i,ichr,ireg
CHARACTER*1 onech,crc(4),creg(4)
EQUIVALENCE (creg,ireg)
ireg=0
creg(ib1)=crc(ib1)          Here is where the character is folded into the register.
creg(ib2)=char(ieor(ichar(crc(ib2)),ichar(onech)))
do 11 i=1,8                 Here is where 8 one-bit shifts, and some XORs with the gen-
    ichr=ichar(creg(ib2))       erator polynomial, are done.
    ireg=ireg+ireg
    creg(ib3)=char(0)
    if(ichr.gt.127)ireg=ieor(ireg,4129)
enddo 11
icrc1=ireg
return
END
```

Now look at `icrc`. There are two parts to understand, how it builds a table when it initializes, and how it uses that table later on. Go back to thinking about a character's bits being shifted into the CRC register from the least significant end. The key observation is that while 8 bits are being shifted into the register's low end, all the generator zapping is being determined by the bits already in the high end. Since XOR is commutative and associative, all we need is a table of the result of all this zapping, for each of 256 possible high-bit configurations. Then we can play catch-up and XOR an input character into the result of a lookup into this table. The routine makes repeated use of an equivalenced 4-byte integer and 4-byte

character array to get at different 8-bit chunks.The only other content to icrc is the construction at initialization time of an 8-bit bit-reverse table from the 4-bit table stored in it, and the logic associated with doing the bit reversals. References [4-6] give further details on table-driven CRC computations.

```
      FUNCTION icrc(crc,bufptr,len,jinit,jrev)
      INTEGER icrc,jinit,jrev,len
      CHARACTER*1 bufptr(*),crc(2)
C     USES icrc1
        Computes a 16-bit Cyclic Redundancy Check for an array bufptr of length len bytes,
        using any of several conventions as determined by the settings of jinit and jrev (see
        accompanying table). The result is returned both as an integer icrc and as a 2-byte array
        crc. If jinit is negative, then crc is used on input to initialize the remainder register, in
        effect concatenating bufptr to the previous call.
      INTEGER ich,init,ireg,j,icrctb(0:255),it(0:15),icrc1,ib1,ib2,ib3
      CHARACTER*1 creg(4),rchr(0:255)
      SAVE icrctb,rchr,init,it,ib1,ib2,ib3
      EQUIVALENCE (creg,ireg)       Used to get at the 4 bytes in an integer.
      DATA it/0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15/, init /0/
        Table of 4-bit bit-reverses, and flag for initialization.
      if (init.eq.0) then           Do we need to initialize tables?
          init=1
          ireg=256*(256*ichar('3')+ichar('2'))+ichar('1')
          do 11 j=1,4                       Figure out which component of creg addresses which
              if (creg(j).eq.'1') ib1=j       byte of ireg.
              if (creg(j).eq.'2') ib2=j
              if (creg(j).eq.'3') ib3=j
          enddo 11
          do 12 j=0,255             The two tables are: CRCs of all characters, and bit-reverses
              ireg=j*256                of all characters.
              icrctb(j)=icrc1(creg,char(0),ib1,ib2,ib3)
              ich=it(mod(j,16))*16+it(j/16)
              rchr(j)=char(ich)
          enddo 12
      endif
      if (jinit.ge.0) then          Initialize the remainder register.
          crc(1)=char(jinit)
          crc(2)=char(jinit)
      else if (jrev.lt.0) then      If not initializing, do we reverse the register?
          ich=ichar(crc(1))
          crc(1)=rchr(ichar(crc(2)))
          crc(2)=rchr(ich)
      endif
      do 13 j=1,len                 Main loop over the characters in the array.
          ich=ichar(bufptr(j))
          if(jrev.lt.0)ich=ichar(rchr(ich))
          ireg=icrctb(ieor(ich,ichar(crc(2))))
          crc(2)=char(ieor(ichar(creg(ib2)),ichar(crc(1))))
          crc(1)=creg(ib1)
      enddo 13
      if (jrev.ge.0) then           Do we need to reverse the output?
          creg(ib1)=crc(1)
          creg(ib2)=crc(2)
      else
          creg(ib2)=rchr(ichar(crc(1)))
          creg(ib1)=rchr(ichar(crc(2)))
          crc(1)=creg(ib1)
          crc(2)=creg(ib2)
      endif
      icrc=ireg
      return
      END
```

What if you need a 32-bit checksum? For a true 32-bit CRC, you will need to rewrite the routines given to work with a longer generating polynomial. For example, $x^{32}+x^7+x^5+x^3+x^2+x+1$ is primitive modulo 2, and has nonleading, nonzero bits only in its least significant byte (which makes for some simplification). The idea of table lookup on only the most significant byte of the CRC register goes through unchanged. Pay attention to the fact that `FORTRAN` does not have unsigned integers, so half of your CRCs will appear to be negative in integer format.

If you do not care about the M-consecutive bit property of the checksum, but rather only need a statistically random 32 bits, then you can use `icrc` as given here: Call it once with `jrev` $= 1$ to get 16 bits, and *again* with `jrev` $= -1$ to get another 16 bits. The internal bit reversals make these two 16-bit CRCs in effect totally independent of each other.

Other Kinds of Checksums

Quite different from CRCs are the various techniques used to append a decimal "check digit" to numbers that are handled by human beings (e.g., typed into a computer). Check digits need to be proof against the kinds of highly structured errors that humans tend to make, such as transposing consecutive digits. Wagner and Putter [7] give an interesting introduction to this subject, including specific algorithms.

Checksums now in widespread use vary from fair to poor. The 10-digit ISBN (International Standard Book Number) that you find on most books, including this one, uses the check equation

$$10d_1 + 9d_2 + 8d_3 + \cdots + 2d_9 + d_{10} = 0 \pmod{11} \tag{20.3.1}$$

where d_{10} is the right-hand check digit. The character "X" is used to represent a check digit value of 10. Another popular scheme is the so-called "IBM check," often used for account numbers (including, e.g., MasterCard). Here, the check equation is

$$2\#d_1 + d_2 + 2\#d_3 + d_4 + \cdots = 0 \pmod{10} \tag{20.3.2}$$

where $2\#d$ means, "multiply d by two and add the resulting decimal digits." United States banks code checks with a 9-digit processing number whose check equation is

$$3a_1 + 7a_2 + a_3 + 3a_4 + 7a_5 + a_6 + 3a_7 + 7a_8 + a_9 = 0 \pmod{10} \tag{20.3.3}$$

The bar code put on many envelopes by the U.S. Postal Service is decoded by removing the single tall marker bars at each end, and breaking the remaining bars into 6 or 10 groups of five. In each group the five bars signify (from left to right) the values 7,4,2,1,0. Exactly two of them will be tall. Their sum is the represented digit, except that zero is represented as $7+4$. The 5- or 9-digit Zip Code is followed by a check digit, with the check equation

$$\sum d_i = 0 \pmod{10} \tag{20.3.4}$$

None of these schemes is close to optimal. An elegant scheme due to Verhoeff is described in [7]. The underlying idea is to use the ten-element *dihedral group* D_5,

which corresponds to the symmetries of a pentagon, instead of the cyclic group of the integers modulo 10. The check equation is

$$a_1 * f(a_2) * f^2(a_3) * \cdots * f^{n-1}(a_n) = 0 \tag{20.3.5}$$

where $*$ is (noncommutative) multiplication in D_5, and f^i denotes the ith iteration of a certain fixed permutation. Verhoeff's method finds *all* single errors in a string, and *all* adjacent transpositions. It also finds about 95% of twin errors ($aa \rightarrow bb$), jump transpositions ($acb \rightarrow bca$), and jump twin errors ($aca \rightarrow bcb$). Here is an implementation:

```
      LOGICAL FUNCTION decchk(string,n,ch)
      INTEGER n
      CHARACTER string*(*),ch*1
          Decimal check digit computation or verification. Returns as ch a check digit for appending
          to string(1:n), that is, for storing into string(n+1:n+1). In this mode, ignore the
          returned logical value. If string(1:n) already ends with a check digit (string(n:n)),
          returns the function value .true. if the check digit is valid, otherwise .false. In this
          mode, ignore the returned value of ch. Note that string and ch contain ASCII characters
          corresponding to the digits 0-9, not byte values in that range. Other ASCII characters are
          allowed in string, and are ignored in calculating the check digit.
      INTEGER ij(10,10),ip(10,8),i,j,k,m
      SAVE ij,ip                                Group multiplication and permutation tables.
      DATA ip/0,1,2,3,4,5,6,7,8,9,1,5,7,6,2,8,3,0,9,4,
     *  5,8,0,3,7,9,6,1,4,2,8,9,1,6,0,4,3,5,2,7,9,4,5,3,1,2,6,8,7,0,
     *  4,2,8,6,5,7,3,9,0,1,2,7,9,3,8,0,6,4,1,5,7,0,4,6,9,1,3,2,5,8/,
     *  ij/0,1,2,3,4,5,6,7,8,9,1,2,3,4,0,9,5,6,7,8,2,3,4,0,1,8,9,5,6,
     *  7,3,4,0,1,2,7,8,9,5,6,4,0,1,2,3,6,7,8,9,5,5,6,7,8,9,0,1,2,3,
     *  4,6,7,8,9,5,4,0,1,2,3,7,8,9,5,6,3,4,0,1,2,8,9,5,6,7,2,3,4,0,
     *  1,9,5,6,7,8,1,2,3,4,0/
      k=0
      m=0
      do 11 j=1,n                               Look at successive characters.
          i=ichar(string(j:j))
          if (i.ge.48.and.i.le.57)then          Ignore everything except digits.
              k=ij(k+1,ip(mod(i+2,10)+1,mod(m,8)+1)+1)
              m=m+1
          endif
      enddo 11
      decchk=(k.eq.0)
      do 12 i=0,9                               Find which appended digit will check properly.
          if (ij(k+1,ip(i+1,mod(m,8)+1)+1).eq.0) goto 1
      enddo 12
1     ch=char(i+48)                             Convert to ASCII.
      return
      end
```

CITED REFERENCES AND FURTHER READING:

McNamara, J.E. 1982, *Technical Aspects of Data Communication*, 2nd ed. (Bedford, MA: Digital Press). [1]

da Cruz, F. 1987, *Kermit, A File Transfer Protocol* (Bedford, MA: Digital Press). [2]

Morse, G. 1986, *Byte*, vol. 11, pp. 115–124 (September). [3]

LeVan, J. 1987, *Byte*, vol. 12, pp. 339–341 (November). [4]

Sarwate, D.V. 1988, *Communications of the ACM*, vol. 31, pp. 1008–1013. [5]

Griffiths, G., and Stones, G.C. 1987, *Communications of the ACM*, vol. 30, pp. 617–620. [6]

Wagner, N.R., and Putter, P.S. 1989, *Communications of the ACM*, vol. 32, pp. 106–110. [7]

20.4 Huffman Coding and Compression of Data

A lossless data compression algorithm takes a string of symbols (typically ASCII characters or bytes) and translates it *reversibly* into another string, one that is *on the average* of shorter length. The words "on the average" are crucial; it is obvious that no reversible algorithm can make all strings shorter — there just aren't enough short strings to be in one-to-one correspondence with longer strings. Compression algorithms are possible only when, on the input side, some strings, or some input symbols, are more common than others. These can then be encoded in fewer bits than rarer input strings or symbols, giving a net average gain.

There exist many, quite different, compression techniques, corresponding to different ways of detecting and using departures from equiprobability in input strings. In this section and the next we shall consider only *variable length codes* with *defined word* inputs. In these, the input is sliced into fixed units, for example ASCII characters, while the corresponding output comes in chunks of variable size. The simplest such method is Huffman coding [1], discussed in this section. Another example, *arithmetic compression*, is discussed in §20.5.

At the opposite extreme from defined-word, variable length codes are schemes that divide up the *input* into units of variable length (words or phrases of English text, for example) and then transmit these, often with a fixed-length output code. The most widely used code of this type is the Ziv-Lempel code [2]. References [3-6] give the flavor of some other compression techniques, with references to the large literature.

The idea behind Huffman coding is simply to use shorter bit patterns for more common characters. We can make this idea quantitative by considering the concept of *entropy*. Suppose the input alphabet has N_{ch} characters, and that these occur in the input string with respective probabilities p_i, $i = 1, \ldots, N_{ch}$, so that $\sum p_i = 1$. Then the fundamental theorem of information theory says that strings consisting of independently random sequences of these characters (a conservative, but not always realistic assumption) require, on the average, at least

$$H = -\sum p_i \log_2 p_i \tag{20.4.1}$$

bits per character. Here H is the entropy of the probability distribution. Moreover, coding schemes exist which approach the bound arbitrarily closely. For the case of equiprobable characters, with all $p_i = 1/N_{ch}$, one easily sees that $H = \log_2 N_{ch}$, which is the case of no compression at all. Any other set of p_i's gives a smaller entropy, allowing some useful compression.

Notice that the bound of (20.4.1) would be achieved if we could encode character i with a code of length $L_i = -\log_2 p_i$ bits: Equation (20.4.1) would then be the average $\sum p_i L_i$. The trouble with such a scheme is that $-\log_2 p_i$ is not generally an integer. How can we encode the letter "Q" in 5.32 bits? Huffman coding makes a stab at this by, in effect, approximating all the probabilities p_i by integer powers of 1/2, so that all the L_i's are integral. If all the p_i's are in fact of this form, then a Huffman code does achieve the entropy bound H.

The construction of a Huffman code is best illustrated by example. Imagine a language, Vowellish, with the $N_{ch} = 5$ character alphabet A, E, I, O, and U, occurring with the respective probabilities 0.12, 0.42, 0.09, 0.30, and 0.07. Then the construction of a Huffman code for Vowellish is accomplished in the following table:

20.4 Huffman Coding and Compression of Data

A lossless data compression algorithm takes a string of symbols (typically ASCII characters or bytes) and translates it *reversibly* into another string, one that is *on the average* of shorter length. The words "on the average" are crucial; it is obvious that no reversible algorithm can make all strings shorter — there just aren't enough short strings to be in one-to-one correspondence with longer strings. Compression algorithms are possible only when, on the input side, some strings, or some input symbols, are more common than others. These can then be encoded in fewer bits than rarer input strings or symbols, giving a net average gain.

There exist many, quite different, compression techniques, corresponding to different ways of detecting and using departures from equiprobability in input strings. In this section and the next we shall consider only *variable length codes* with *defined word* inputs. In these, the input is sliced into fixed units, for example ASCII characters, while the corresponding output comes in chunks of variable size. The simplest such method is Huffman coding [1], discussed in this section. Another example, *arithmetic compression*, is discussed in §20.5.

At the opposite extreme from defined-word, variable length codes are schemes that divide up the *input* into units of variable length (words or phrases of English text, for example) and then transmit these, often with a fixed-length output code. The most widely used code of this type is the Ziv-Lempel code [2]. References [3-6] give the flavor of some other compression techniques, with references to the large literature.

The idea behind Huffman coding is simply to use shorter bit patterns for more common characters. We can make this idea quantitative by considering the concept of *entropy*. Suppose the input alphabet has N_{ch} characters, and that these occur in the input string with respective probabilities p_i, $i = 1, \ldots, N_{ch}$, so that $\sum p_i = 1$. Then the fundamental theorem of information theory says that strings consisting of independently random sequences of these characters (a conservative, but not always realistic assumption) require, on the average, at least

$$H = -\sum p_i \log_2 p_i \tag{20.4.1}$$

bits per character. Here H is the entropy of the probability distribution. Moreover, coding schemes exist which approach the bound arbitrarily closely. For the case of equiprobable characters, with all $p_i = 1/N_{ch}$, one easily sees that $H = \log_2 N_{ch}$, which is the case of no compression at all. Any other set of p_i's gives a smaller entropy, allowing some useful compression.

Notice that the bound of (20.4.1) would be achieved if we could encode character i with a code of length $L_i = -\log_2 p_i$ bits: Equation (20.4.1) would then be the average $\sum p_i L_i$. The trouble with such a scheme is that $-\log_2 p_i$ is not generally an integer. How can we encode the letter "Q" in 5.32 bits? Huffman coding makes a stab at this by, in effect, approximating all the probabilities p_i by integer powers of 1/2, so that all the L_i's are integral. If all the p_i's are in fact of this form, then a Huffman code does achieve the entropy bound H.

The construction of a Huffman code is best illustrated by example. Imagine a language, Vowellish, with the $N_{ch} = 5$ character alphabet A, E, I, O, and U, occurring with the respective probabilities 0.12, 0.42, 0.09, 0.30, and 0.07. Then the construction of a Huffman code for Vowellish is accomplished in the following table:

Node	Stage:	1	2	3	4	5
1	A:	0.12	0.12 ■			
2	E:	0.42	0.42	0.42	0.42 ■	
3	I:	0.09 ■				
4	O:	0.30	0.30	0.30 ■		
5	U:	0.07 ■				
6		UI:	0.16 ■			
7			AUI:	0.28 ■		
8				AUIO:	0.58 ■	
9					EAUIO:	1.00

Here is how it works, proceeding in sequence through N_{ch} stages, represented by the columns of the table. The first stage starts with N_{ch} nodes, one for each letter of the alphabet, containing their respective relative frequencies. At each stage, the two smallest probabilities are found, summed to make a new node, and then dropped from the list of active nodes. (A "block" denotes the stage where a node is dropped.) All active nodes (including the new composite) are then carried over to the next stage (column). In the table, the names assigned to new nodes (e.g., AUI) are inconsequential. In the example shown, it happens that (after stage 1) the two smallest nodes are always an original node and a composite one; this need not be true in general: The two smallest probabilities might be both original nodes, or both composites, or one of each. At the last stage, all nodes will have been collected into one grand composite of total probability 1.

Now, to see the code, you redraw the data in the above table as a tree (Figure 20.4.1). As shown, each node of the tree corresponds to a node (row) in the table, indicated by the integer to its left and probability value to its right. Terminal nodes, so called, are shown as circles; these are single alphabetic characters. The branches of the tree are labeled 0 and 1. The code for a character is the sequence of zeros and ones that lead to it, from the top down. For example, E is simply 0, while U is 1010.

Any string of zeros and ones can now be decoded into an alphabetic sequence. Consider, for example, the string 1011111010. Starting at the top of the tree we descend through 1011 to I, the first character. Since we have reached a terminal node, we reset to the top of the tree, next descending through 11 to O. Finally 1010 gives U. The string thus decodes to IOU.

These ideas are embodied in the following routines. Input to the first routine `hufmak` is an integer vector of the frequency of occurrence of the `nchin` $\equiv N_{ch}$ alphabetic characters, i.e., a set of integers proportional to the p_i's. `hufmak`, along with `hufapp`, which it calls, performs the construction of the above table, and also the tree of Figure 20.4.1. The routine utilizes a heap structure (see §8.3) for efficiency; for a detailed description, see Sedgewick [7].

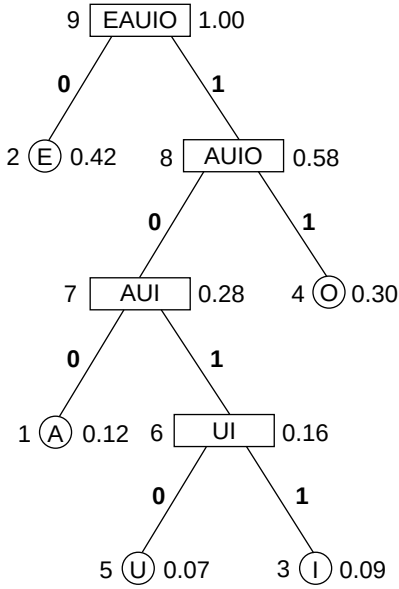


Figure 20.4.1. Huffman code for the fictitious language Vowellish, in tree form. A letter (A, E, I, O, or U) is encoded or decoded by traversing the tree from the top down; the code is the sequence of 0's and 1's on the branches. The value to the right of each node is its probability; to the left, its node number in the accompanying table.

```
      SUBROUTINE hufmak(nfreq,nchin,ilong,nlong)
      INTEGER ilong,nchin,nlong,nfreq(nchin),MC,MQ
      PARAMETER (MC=512,MQ=2*MC-1)
C     USES hufapp
        Given the frequency of occurrence table nfreq(1:nchin) of nchin characters, construct
        in the common block /hufcom/ the Huffman code. Returned values ilong and nlong
        are the character number that produced the longest code symbol, and the length of that
        symbol. You should check that nlong is not larger than your machine's word length.
      INTEGER ibit,j,k,n,nch,node,nodemx,nused,ibset,index(MQ),
     *     iup(MQ),icod(MQ),left(MQ),iright(MQ),ncod(MQ),nprob(MQ)
      COMMON /hufcom/ icod,ncod,nprob,left,iright,nch,nodemx
      SAVE /hufcom/
      nch=nchin                         Initialization.
      nused=0
      do 11 j=1,nch
        nprob(j)=nfreq(j)
        icod(j)=0
        ncod(j)=0
        if(nfreq(j).ne.0)then
          nused=nused+1
          index(nused)=j
        endif
      enddo 11
      do 12 j=nused,1,-1                Sort nprob into a heap structure in index.
        call hufapp(index,nprob,nused,j)
      enddo 12
      k=nch
1     if(nused.gt.1)then                Combine heap nodes, remaking the heap at each stage.
      node=index(1)
        index(1)=index(nused)
        nused=nused-1
        call hufapp(index,nprob,nused,1)
        k=k+1
```

```
        nprob(k)=nprob(index(1))+nprob(node)
        left(k)=node              Store left and right children of a node.
        iright(k)=index(1)
        iup(index(1)) = -k        Indicate whether a node is a left or right child of its parent.
        iup(node)=k
        index(1)=k
        call hufapp(index,nprob,nused,1)
    goto 1
    endif
    nodemx=k
    iup(nodemx)=0
    do 13 j=1,nch                 Make the Huffman code from the tree.
        if(nprob(j).ne.0)then
            n=0
            ibit=0
            node=iup(j)
2           if(node.ne.0)then
                if(node.lt.0)then
                    n=ibset(n,ibit)
                    node = -node
                endif
                node=iup(node)
                ibit=ibit+1
            goto 2
            endif
            icod(j)=n
            ncod(j)=ibit
        endif
    enddo 13
    nlong=0
    do 14 j=1,nch
        if(ncod(j).gt.nlong)then
            nlong=ncod(j)
            ilong=j-1
        endif
    enddo 14
    return
    END



    SUBROUTINE hufapp(index,nprob,m,l)
    INTEGER m,l,MC,MQ
    PARAMETER (MC=512,MQ=2*MC-1)
    INTEGER index(MQ),nprob(MQ)
       Used by hufmak to maintain a heap structure in the array index(1:l).
    INTEGER i,j,k,n
    n=m
    i=l
    k=index(i)
2   if(i.le.n/2)then
        j=i+i
        if (j.lt.n.and.nprob(index(j)).gt.nprob(index(j+1))) j=j+1
        if (nprob(k).le.nprob(index(j))) goto 3
        index(i)=index(j)
        i=j
    goto 2
    endif
3   index(i)=k
    return
    END
```

Once the code is constructed, one encodes a string of characters by repeated calls to hufenc, which simply does a table lookup of the code and appends it to the output message.

```
      SUBROUTINE hufenc(ich,code,lcode,nb)
      INTEGER ich,lcode,nb,MC,MQ
      PARAMETER (MC=512,MQ=2*MC-1)
        Huffman encode the single character ich (in the range 0..nch-1), write the result to the
        character array code(1:lcode) starting at bit nb (whose smallest valid value is zero),
        and increment nb appropriately. This routine is called repeatedly to encode consecutive
        characters in a message, but must be preceded by a single initializing call to hufmak.
      INTEGER k,l,n,nc,nch,nodemx,ntmp,ibset
      INTEGER icod(MQ),left(MQ),iright(MQ),ncod(MQ),nprob(MQ)
      LOGICAL btest
      CHARACTER*1 code(*)
      COMMON /hufcom/ icod,ncod,nprob,left,iright,nch,nodemx
      SAVE /hufcom/
      k=ich+1                          Convert character range 0..nch-1 to array index range 1..nch.
      if(k.gt.nch.or.k.lt.1)pause 'ich out of range in hufenc.'
      do 11 n=ncod(k),1,-1             Loop over the bits in the stored Huffman code for ich.
        nc=nb/8+1
        if (nc.gt.lcode) pause 'lcode too small in hufenc.'
        l=mod(nb,8)
        if (l.eq.0) code(nc)=char(0)
        if(btest(icod(k),n-1))then  Set appropriate bits in code.
          ntmp=ibset(ichar(code(nc)),l)
          code(nc)=char(ntmp)
        endif
        nb=nb+1
      enddo 11
      return
      END
```

Decoding a Huffman-encoded message is slightly more complicated. The coding tree must be traversed from the top down, using up a variable number of bits:

```
      SUBROUTINE hufdec(ich,code,lcode,nb)
      INTEGER ich,lcode,nb,MC,MQ
      PARAMETER (MC=512,MQ=2*MC-1)
        Starting at bit number nb in the character array code(1:lcode), use the Huffman code
        stored in common block /hufcom/ to decode a single character (returned as ich in the
        range 0..nch-1) and increment nb appropriately. Repeated calls, starting with nb = 0
        will return successive characters in a compressed message. The returned value ich=nch
        indicates end-of-message. This routine must be preceded by a single initializing call to
        hufmak.
        Parameters: MC is the maximum value of nch, the input alphabet size.
      INTEGER l,nc,nch,node,nodemx
      INTEGER icod(MQ),left(MQ),iright(MQ),ncod(MQ),nprob(MQ)
      LOGICAL btest
      CHARACTER*1 code(lcode)
      COMMON /hufcom/ icod,ncod,nprob,left,iright,nch,nodemx
      SAVE /hufcom/
      node=nodemx                      Set node to the top of the decoding tree.
1     continue                         Loop until a valid character is obtained.
        nc=nb/8+1
        if (nc.gt.lcode)then           Ran out of input; with ich=nch indicating end of message.
          ich=nch
          return
        endif
        l=mod(nb,8)                    Now decoding this bit.
        nb=nb+1
```

```
    if(btest(ichar(code(nc)),1))then  Branch left or right in tree, depending on its
        node=iright(node)                 value.
    else
        node=left(node)
    endif
    if(node.le.nch)then       If we reach a terminal node, we have a complete character
        ich=node-1                and can return.
        return
    endif
goto 1
END
```

For simplicity, `hufdec` quits when it runs out of code bytes; if your coded message is not an integral number of bytes, and if N_{ch} is less than 256, `hufdec` can return a spurious final character or two, decoded from the spurious trailing bits in your last code byte. If you have independent knowledge of the number of characters sent, you can readily discard these. Otherwise, you can fix this behavior by providing a bit, not byte, count, and modifying the routine accordingly. (When N_{ch} is 256 or larger, `hufdec` will normally run out of code in the middle of a spurious character, and it will be discarded.)

Run-Length Encoding

For the compression of highly correlated bit-streams (for example the black or white values along a facsimile scan line), Huffman compression is often combined with *run-length encoding*: Instead of sending each bit, the input stream is converted to a series of integers indicating how many consecutive bits have the same value. These integers are then Huffman-compressed. The Group 3 CCITT facsimile standard functions in this manner, with a fixed, immutable, Huffman code, optimized for a set of eight standard documents [8,9].

CITED REFERENCES AND FURTHER READING:

Gallager, R.G. 1968, *Information Theory and Reliable Communication* (New York: Wiley).

Hamming, R.W. 1980, *Coding and Information Theory* (Englewood Cliffs, NJ: Prentice-Hall).

Storer, J.A. 1988, *Data Compression: Methods and Theory* (Rockville, MD: Computer Science Press).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books).

Huffman, D.A. 1952, *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098–1101. [1]

Ziv, J., and Lempel, A. 1978, *IEEE Transactions on Information Theory*, vol. IT-24, pp. 530–536. [2]

Cleary, J.G., and Witten, I.H. 1984, *IEEE Transactions on Communications*, vol. COM-32, pp. 396–402. [3]

Welch, T.A. 1984, *Computer*, vol. 17, no. 6, pp. 8–19. [4]

Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. 1986, *Communications of the ACM*, vol. 29, pp. 320–330. [5]

Jones, D.W. 1988, *Communications of the ACM*, vol. 31, pp. 996–1007. [6]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 22. [7]

Hunter, R., and Robinson, A.H. 1980, *Proceedings of the IEEE*, vol. 68, pp. 854–867. [8]

Marking, M.P. 1990, *The C Users' Journal*, vol. 8, no. 6, pp. 45–54. [9]

20.5 Arithmetic Coding

We saw in the previous section that a perfect (entropy-bounded) coding scheme would use $L_i = -\log_2 p_i$ bits to encode character i (in the range $1 \le i \le N_{ch}$), if p_i is its probability of occurrence. Huffman coding gives a way of rounding the L_i's to close integer values and constructing a code with those lengths. *Arithmetic coding* [1], which we now discuss, actually does manage to encode characters using noninteger numbers of bits! It also provides a convenient way to output the result not as a stream of bits, but as a stream of symbols in any desired radix. This latter property is particularly useful if you want, e.g., to convert data from bytes (radix 256) to printable ASCII characters (radix 94), or to case-independent alphanumeric sequences containing only A-Z and 0-9 (radix 36).

In arithmetic coding, an input message of any length is represented as a real number R in the range $0 \le R < 1$. The longer the message, the more precision required of R. This is best illustrated by an example, so let us return to the fictitious language, Vowellish, of the previous section. Recall that Vowellish has a 5 character alphabet (A, E, I, O, U), with occurrence probabilities 0.12, 0.42, 0.09, 0.30, and 0.07, respectively. Figure 20.5.1 shows how a message beginning "IOU" is encoded: The interval $[0, 1)$ is divided into segments corresponding to the 5 alphabetical characters; the length of a segment is the corresponding p_i. We see that the first message character, "I", narrows the range of R to $0.37 \le R < 0.46$. This interval is now subdivided into five subintervals, again with lengths proportional to the p_i's. The second message character, "O", narrows the range of R to $0.3763 \le R < 0.4033$. The "U" character further narrows the range to $0.37630 \le R < 0.37819$. *Any* value of R in this range can be sent as encoding "IOU". In particular, the binary fraction $.011000001$ is in this range, so "IOU" can be sent in 9 bits. (Huffman coding took 10 bits for this example, see §20.4.)

Of course there is the problem of knowing when to stop decoding. The fraction .011000001 represents not simply "IOU," but "IOU. . .," where the ellipses represent an infinite string of successor characters. To resolve this ambiguity, arithmetic coding generally assumes the existence of a special $N_{ch} + 1$th character, EOM (end of message), which occurs only once at the end of the input. Since EOM has a low probability of occurrence, it gets allocated only a very tiny piece of the number line.

In the above example, we gave R as a binary fraction. We could just as well have output it in any other radix, e.g., base 94 or base 36, whatever is convenient for the anticipated storage or communication channel.

You might wonder how one deals with the seemingly incredible precision required of R for a long message. The answer is that R is never actually represented all at once. At any give stage we have upper and lower bounds for R represented as a finite number of digits in the output radix. As digits of the upper and lower bounds become identical, we can left-shift them away and bring in new digits at the low-significance end. The routines below have a parameter `NWK` for the number of working digits to keep around. This must be large enough to make the chance of an accidental degeneracy vanishingly small. (The routines signal if a degeneracy ever occurs.) Since the process of discarding old digits and bringing in new ones is performed identically on encoding and decoding, everything stays synchronized.

20.5 Arithmetic Coding

We saw in the previous section that a perfect (entropy-bounded) coding scheme would use $L_i = -\log_2 p_i$ bits to encode character i (in the range $1 \le i \le N_{ch}$), if p_i is its probability of occurrence. Huffman coding gives a way of rounding the L_i's to close integer values and constructing a code with those lengths. *Arithmetic coding* [1], which we now discuss, actually does manage to encode characters using noninteger numbers of bits! It also provides a convenient way to output the result not as a stream of bits, but as a stream of symbols in any desired radix. This latter property is particularly useful if you want, e.g., to convert data from bytes (radix 256) to printable ASCII characters (radix 94), or to case-independent alphanumeric sequences containing only A-Z and 0-9 (radix 36).

In arithmetic coding, an input message of any length is represented as a real number R in the range $0 \le R < 1$. The longer the message, the more precision required of R. This is best illustrated by an example, so let us return to the fictitious language, Vowellish, of the previous section. Recall that Vowellish has a 5 character alphabet (A, E, I, O, U), with occurrence probabilities 0.12, 0.42, 0.09, 0.30, and 0.07, respectively. Figure 20.5.1 shows how a message beginning "IOU" is encoded: The interval $[0, 1)$ is divided into segments corresponding to the 5 alphabetical characters; the length of a segment is the corresponding p_i. We see that the first message character, "I", narrows the range of R to $0.37 \le R < 0.46$. This interval is now subdivided into five subintervals, again with lengths proportional to the p_i's. The second message character, "O", narrows the range of R to $0.3763 \le R < 0.4033$. The "U" character further narrows the range to $0.37630 \le R < 0.37819$. *Any* value of R in this range can be sent as encoding "IOU". In particular, the binary fraction $.011000001$ is in this range, so "IOU" can be sent in 9 bits. (Huffman coding took 10 bits for this example, see §20.4.)

Of course there is the problem of knowing when to stop decoding. The fraction .011000001 represents not simply "IOU," but "IOU. . .," where the ellipses represent an infinite string of successor characters. To resolve this ambiguity, arithmetic coding generally assumes the existence of a special $N_{ch} + 1$th character, EOM (end of message), which occurs only once at the end of the input. Since EOM has a low probability of occurrence, it gets allocated only a very tiny piece of the number line.

In the above example, we gave R as a binary fraction. We could just as well have output it in any other radix, e.g., base 94 or base 36, whatever is convenient for the anticipated storage or communication channel.

You might wonder how one deals with the seemingly incredible precision required of R for a long message. The answer is that R is never actually represented all at once. At any give stage we have upper and lower bounds for R represented as a finite number of digits in the output radix. As digits of the upper and lower bounds become identical, we can left-shift them away and bring in new digits at the low-significance end. The routines below have a parameter `NWK` for the number of working digits to keep around. This must be large enough to make the chance of an accidental degeneracy vanishingly small. (The routines signal if a degeneracy ever occurs.) Since the process of discarding old digits and bringing in new ones is performed identically on encoding and decoding, everything stays synchronized.

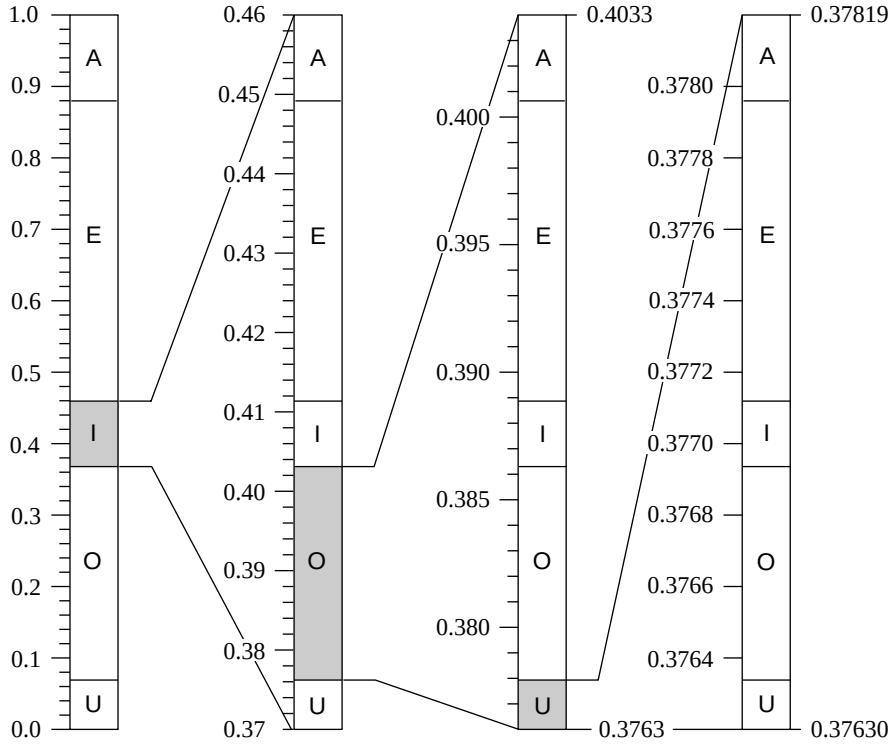


Figure 20.5.1. Arithmetic coding of the message "IOU..." in the fictitious language Vowellish. Successive characters give successively finer subdivisions of the initial interval between 0 and 1. The final value can be output as the digits of a fraction in any desired radix. Note how the subinterval allocated to a character is proportional to its probability of occurrence.

The routine `arcmak` constructs the cumulative frequency distribution table used to partition the interval at each stage. In the principal routine `arcode`, when an interval of size `jdif` is to be partitioned in the proportions of some `n` to some `ntot`, say, then we must compute `(n*jdif)/ntot`. With integer arithmetic, the numerator is likely to overflow; and, unfortunately, an expression like `jdif/(ntot/n)` is not equivalent. In the implementation below, we resort to double precision floating arithmetic for this calculation. Not only is this inefficient, but different roundoff errors can (albeit very rarely) make different machines encode differently, though any one type of machine will decode exactly what it encoded, since identical roundoff errors occur in the two processes. For serious use, one needs to replace this floating calculation with an integer computation in a double register (not available to the `FORTRAN` programmer).

The internally set variable `minint`, which is the minimum allowed number of discrete steps between the upper and lower bounds, determines when new low-significance digits are added. `minint` must be large enough to provide resolution of all the input characters. That is, we must have $p_i \times \texttt{minint} > 1$ for all i. A value of $100N_{ch}$, or $1.1/\min p_i$, whichever is larger, is generally adequate. However, for safety, the routine below takes `minint` to be as large as possible, with the product `minint*nradd` just smaller than overflow. This results in some time inefficiency, and in a few unnecessary characters being output at the end of a message. You can

decrease minint if you want to live closer to the edge.

A final safety feature in arcmak is its refusal to believe zero values in the table nfreq; a 0 is treated as if it were a 1. If this were not done, the occurrence in a message of a single character whose nfreq entry is zero would result in scrambling the entire rest of the message. If you want to live dangerously, with a very slightly more efficient coding, you can delete the max(,1) operation.

```
      SUBROUTINE arcmak(nfreq,nchh,nradd)
      INTEGER nchh,nradd,nfreq(nchh),MC,NWK,MAXINT
      PARAMETER (MC=512,NWK=20,MAXINT=2147483647)
        Given a table nfreq(1:nchh) of the frequency of occurrence of nchh symbols, and given
        a desired output radix nradd, initialize the cumulative frequency table and other variables
        for arithmetic compression.
        Parameters: MC is largest anticipated value of nchh; NWK is the number of working digits
        (see text); MAXINT is a large positive integer that does not overflow.
      INTEGER j,jdif,minint,nc,nch,nrad,ncum,
     *     ncumfq(MC+2),ilob(NWK),iupb(NWK)
      COMMON /arccom/ ncumfq,iupb,ilob,nch,nrad,minint,jdif,nc,ncum
      SAVE /arccom/
      if(nchh.gt.MC)pause 'MC too small in arcmak'
      if(nradd.gt.256)pause 'nradd may not exceed 256 in arcmak'
      minint=MAXINT/nradd
      nch=nchh
      nrad=nradd
      ncumfq(1)=0
      do 11 j=2,nch+1
        ncumfq(j)=ncumfq(j-1)+max(nfreq(j-1),1)
      enddo 11
      ncumfq(nch+2)=ncumfq(nch+1)+1
      ncum=ncumfq(nch+2)
      return
      END
```

Individual characters in a message are coded or decoded by the routine arcode, which in turn uses the utility arcsum.

```
      SUBROUTINE arcode(ich,code,lcode,lcd,isign)
      INTEGER ich,isign,lcd,lcode,MC,NWK
      CHARACTER*1 code(lcode)
      PARAMETER (MC=512,NWK=20)
C     USES arcsum
        Compress (isign = 1) or decompress (isign = −1) the single character ich into or out
        of the character array code(1:lcode), starting with byte code(lcd) and (if necessary)
        incrementing lcd so that, on return, lcd points to the first unused byte in code. Note
        that this routine saves the result of previous calls until a new byte of code is produced, and
        only then increments lcd. An initializing call with isign=0 is required for each different
        array code. The routine arcmak must have previously been called to initialize the common
        block /arccom/. A call with ich=nch (as set in arcmak) has the reserved meaning "end
        of message."
      INTEGER ihi,j,ja,jdif,jh,jl,k,m,minint,nc,nch,nrad,ilob(NWK),
     *     iupb(NWK),ncumfq(MC+2),ncum,JTRY
      COMMON /arccom/ ncumfq,iupb,ilob,nch,nrad,minint,jdif,nc,ncum
      SAVE /arccom/
        The following statement function is used to calculate (k*j)/m without overflow. Program
        efficiency can be improved by substituting an assembly language routine that does integer
        multiply to a double register.
      JTRY(j,k,m)=int((dble(k)*dble(j))/dble(m))
      if (isign.eq.0) then                Initialize enough digits of the upper and lower bounds.
        jdif=nrad-1
        do 11 j=NWK,1,-1
```

```
            iupb(j)=nrad-1
            ilob(j)=0
            nc=j
            if(jdif.gt.minint)return    Initialization complete.
            jdif=(jdif+1)*nrad-1
        enddo 11
        pause 'NWK too small in arcode'
    else
        if (isign.gt.0) then            If encoding, check for valid input character.
            if(ich.gt.nch.or.ich.lt.0)pause 'bad ich in arcode'
        else                            If decoding, locate the character ich by bisection.
            ja=ichar(code(lcd))-ilob(nc)
            do 12 j=nc+1,NWK
                ja=ja*nrad+(ichar(code(j+lcd-nc))-ilob(j))
            enddo 12
            ich=0
            ihi=nch+1
1           if(ihi-ich.gt.1) then
                m=(ich+ihi)/2
                if (ja.ge.JTRY(jdif,ncumfq(m+1),ncum)) then
                    ich=m
                else
                    ihi=m
                endif
            goto 1
            endif
            if(ich.eq.nch)return        Detected end of message.
        endif
          Following code is common for encoding and decoding. Convert character ich to a new
          subrange [ilob,iupb).
        jh=JTRY(jdif,ncumfq(ich+2),ncum)
        jl=JTRY(jdif,ncumfq(ich+1),ncum)
        jdif=jh-jl
        call arcsum(ilob,iupb,jh,NWK,nrad,nc)
        call arcsum(ilob,ilob,jl,NWK,nrad,nc)       How many leading digits to output
        do 13 j=nc,NWK                                  (if encoding) or skip over?
            if(ich.ne.nch.and.iupb(j).ne.ilob(j))goto 2
            if(lcd.gt.lcode)pause 'lcode too small in arcode'
            if(isign.gt.0) code(lcd)=char(ilob(j))
            lcd=lcd+1
        enddo 13
        return                          Ran out of message. Did someone forget to encode
2       nc=j                                a terminating ncd?
        j=0                             How many digits to shift?
3       if (jdif.lt.minint) then
            j=j+1
            jdif=jdif*nrad
        goto 3
        endif
        if (nc-j.lt.1) pause 'NWK too small in arcode'
        if(j.ne.0)then                  Shift them.
            do 14 k=nc,NWK
                iupb(k-j)=iupb(k)
                ilob(k-j)=ilob(k)
            enddo 14
        endif
        nc=nc-j
        do 15 k=NWK-j+1,NWK
            iupb(k)=0
            ilob(k)=0
        enddo 15
    endif
    return                              Normal return.
    END
```

```
SUBROUTINE arcsum(iin,iout,ja,nwk,nrad,nc)
INTEGER ja,nc,nrad,nwk,iin(*),iout(*)
    Used by arcode. Add the integer ja to the radix nrad multiple-precision integer iin(nc..nwk).
    Return the result in iout(nc..nwk).
INTEGER j,jtmp,karry
karry=0
do 11 j=nwk,nc+1,-1
    jtmp=ja
    ja=ja/nrad
    iout(j)=iin(j)+(jtmp-ja*nrad)+karry
    if (iout(j).ge.nrad) then
        iout(j)=iout(j)-nrad
        karry=1
    else
        karry=0
    endif
enddo 11
iout(nc)=iin(nc)+ja+karry
return
END
```

If radix-changing, rather than compression, is your primary aim (for example to convert an arbitrary file into printable characters) then you are of course free to set all the components of `nfreq` equal, say, to 1.

CITED REFERENCES AND FURTHER READING:

Bell, T.C., Cleary, J.G., and Witten, I.H. 1990, *Text Compression* (Englewood Cliffs, NJ: Prentice-Hall).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books).

Witten, I.H., Neal, R.M., and Cleary, J.G. 1987, *Communications of the ACM*, vol. 30, pp. 520–540. [1]

20.6 Arithmetic at Arbitrary Precision

Let's compute the number π to a couple of thousand decimal places. In doing so, we'll learn some things about multiple precision arithmetic on computers and meet quite an unusual application of the fast Fourier transform (FFT). We'll also develop a set of routines that you can use for other calculations at any desired level of arithmetic precision.

To start with, we need an analytic algorithm for π. Useful algorithms are quadratically convergent, i.e., they double the number of significant digits at each iteration. Quadratically convergent algorithms for π are based on the *AGM (arithmetic geometric mean)* method, which also finds application to the calculation of elliptic integrals (cf. §6.11) and in advanced implementations of the ADI method for elliptic partial differential equations (§19.5). Borwein and Borwein [1] treat this subject, which is beyond our scope here. One of their algorithms for π starts with the initializations

$$\begin{aligned} X_0 &= \sqrt{2} \\ \pi_0 &= 2 + \sqrt{2} \\ Y_0 &= \sqrt[4]{2} \end{aligned} \tag{20.6.1}$$

```
SUBROUTINE arcsum(iin,iout,ja,nwk,nrad,nc)
INTEGER ja,nc,nrad,nwk,iin(*),iout(*)
    Used by arcode. Add the integer ja to the radix nrad multiple-precision integer iin(nc..nwk).
    Return the result in iout(nc..nwk).
INTEGER j,jtmp,karry
karry=0
do 11 j=nwk,nc+1,-1
    jtmp=ja
    ja=ja/nrad
    iout(j)=iin(j)+(jtmp-ja*nrad)+karry
    if (iout(j).ge.nrad) then
        iout(j)=iout(j)-nrad
        karry=1
    else
        karry=0
    endif
enddo 11
iout(nc)=iin(nc)+ja+karry
return
END
```

If radix-changing, rather than compression, is your primary aim (for example to convert an arbitrary file into printable characters) then you are of course free to set all the components of `nfreq` equal, say, to 1.

CITED REFERENCES AND FURTHER READING:

Bell, T.C., Cleary, J.G., and Witten, I.H. 1990, *Text Compression* (Englewood Cliffs, NJ: Prentice-Hall).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books).

Witten, I.H., Neal, R.M., and Cleary, J.G. 1987, *Communications of the ACM*, vol. 30, pp. 520–540. [1]

20.6 Arithmetic at Arbitrary Precision

Let's compute the number π to a couple of thousand decimal places. In doing so, we'll learn some things about multiple precision arithmetic on computers and meet quite an unusual application of the fast Fourier transform (FFT). We'll also develop a set of routines that you can use for other calculations at any desired level of arithmetic precision.

To start with, we need an analytic algorithm for π. Useful algorithms are quadratically convergent, i.e., they double the number of significant digits at each iteration. Quadratically convergent algorithms for π are based on the *AGM (arithmetic geometric mean)* method, which also finds application to the calculation of elliptic integrals (cf. §6.11) and in advanced implementations of the ADI method for elliptic partial differential equations (§19.5). Borwein and Borwein [1] treat this subject, which is beyond our scope here. One of their algorithms for π starts with the initializations

$$\begin{aligned} X_0 &= \sqrt{2} \\ \pi_0 &= 2 + \sqrt{2} \\ Y_0 &= \sqrt[4]{2} \end{aligned} \tag{20.6.1}$$

and then, for $i = 0, 1, \ldots$, repeats the iteration

$$\begin{aligned} X_{i+1} &= \frac{1}{2}\left(\sqrt{X_i} + \frac{1}{\sqrt{X_i}}\right) \\ \pi_{i+1} &= \pi_i \left(\frac{X_{i+1}+1}{Y_i+1}\right) \\ Y_{i+1} &= \frac{Y_i\sqrt{X_{i+1}} + \dfrac{1}{\sqrt{X_{i+1}}}}{Y_i+1} \end{aligned} \tag{20.6.2}$$

The value π emerges as the limit π_∞.

Now, to the question of how to do arithmetic to arbitrary precision: In a high-level language like FORTRAN, a natural choice is to work in radix (base) 256, so that character arrays can be directly interpreted as strings of digits. At the very end of our calculation, we will want to convert our answer to radix 10, but that is essentially a frill for the benefit of human ears, accustomed to the familiar chant, "three point one four one five nine. . . ." For any less frivolous calculation, we would likely never leave base 256 (or the thence trivially reachable hexadecimal, octal, or binary bases).

We will adopt the convention of storing digit strings in the "human" ordering, that is, with the first stored digit in an array being most significant, the last stored digit being least significant. The opposite convention would, of course, also be possible. "Carries," where we need to partition a number larger than 255 into a low-order byte and a high-order carry, present a minor programming annoyance, solved, in the routines below, by the use of FORTRAN's EQUIVALENCE facility, and some initial testing of the order in which bytes are stored in a FORTRAN integer.

It is easy at this point, following Knuth [2], to write a routine for the "fast" arithmetic operations: short addition (adding a single byte to a string), addition, subtraction, short multiplication (multiplying a string by a single byte), short division, ones-complement negation; and a couple of utility operations, copying and left-shifting strings.

```
SUBROUTINE mpops(w,u,v)
CHARACTER*1 w(*),u(*),v(*)
    Multiple precision arithmetic operations done on character strings, interpreted as radix 256
    numbers. This routine collects the simpler operations.
INTEGER i,ireg,j,n,ir,is,iv,ii1,ii2
CHARACTER*1 creg(4)
SAVE ii1,ii2
EQUIVALENCE (ireg,creg)
    It is assumed that with the above equivalence, creg(ii1) addresses the low-order byte of
    ireg, and creg(ii2) addresses the next higher order byte. The values ii1 and ii2 are
    set by an initial call to mpinit.
ENTRY mpinit
    ireg=256*ichar('2')+ichar('1')
    do 11 j=1,4                    Figure out the byte ordering.
        if (creg(j).eq.'1') ii1=j
        if (creg(j).eq.'2') ii2=j
    enddo 11
return
ENTRY mpadd(w,u,v,n)
    Adds the unsigned radix 256 integers u(1:n) and v(1:n) yielding the unsigned integer
    w(1:n+1).
    ireg=0
    do 12 j=n,1,-1
```

```
        ireg=ichar(u(j))+ichar(v(j))+ichar(creg(ii2))
        w(j+1)=creg(ii1)
    enddo 12
    w(1)=creg(ii2)
return
ENTRY mpsub(is,w,u,v,n)
```

Subtracts the unsigned radix 256 integer `v(1:n)` from `u(1:n)` yielding the unsigned integer `w(1:n)`. If the result is negative (wraps around), `is` is returned as -1; otherwise it is returned as 0.

```
    ireg=256
    do 13 j=n,1,-1
        ireg=255+ichar(u(j))-ichar(v(j))+ichar(creg(ii2))
        w(j)=creg(ii1)
    enddo 13
    is=ichar(creg(ii2))-1
return
ENTRY mpsad(w,u,n,iv)
```

Short addition: the integer `iv` (in the range $0 \leq$ `iv` ≤ 255) is added to the unsigned radix 256 integer `u(1:n)`, yielding `w(1:n+1)`.

```
    ireg=256*iv
    do 14 j=n,1,-1
        ireg=ichar(u(j))+ichar(creg(ii2))
        w(j+1)=creg(ii1)
    enddo 14
    w(1)=creg(ii2)
return
ENTRY mpsmu(w,u,n,iv)
```

Short multiplication: the unsigned radix 256 integer `u(1:n)` is multiplied by the integer `iv` (in the range $0 \leq$ `iv` ≤ 255), yielding `w(1:n+1)`.

```
    ireg=0
    do 15 j=n,1,-1
        ireg=ichar(u(j))*iv+ichar(creg(ii2))
        w(j+1)=creg(ii1)
    enddo 15
    w(1)=creg(ii2)
return
ENTRY mpsdv(w,u,n,iv,ir)
```

Short division: the unsigned radix 256 integer `u(1:n)` is divided by the integer `iv` (in the range $0 \leq$ `iv` ≤ 255), yielding a quotient `w(1:n)` and a remainder `ir` (with $0 \leq$ `ir` ≤ 255).

```
    ir=0
    do 16 j=1,n
        i=256*ir+ichar(u(j))
        w(j)=char(i/iv)
        ir=mod(i,iv)
    enddo 16
return
ENTRY mpneg(u,n)
```

Ones-complement negate the unsigned radix 256 integer `u(1:n)`.

```
    ireg=256
    do 17 j=n,1,-1
        ireg=255-ichar(u(j))+ichar(creg(ii2))
        u(j)=creg(ii1)
    enddo 17
return
ENTRY mpmov(u,v,n)
```

Move `v(1:n)` onto `u(1:n)`.

```
    do 18 j=1,n
        u(j)=v(j)
    enddo 18
return
ENTRY mplsh(u,n)
```

Left shift `u(2..n+1)` onto `u(1:n)`.

```
    do 19 j=1,n
        u(j)=u(j+1)
```

```
    enddo 19
return
END
```

Full multiplication of two digit strings, if done by the traditional hand method, is not a fast operation: In multiplying two strings of length N, the multiplicand would be short-multiplied in turn by each byte of the multiplier, requiring $O(N^2)$ operations in all. We will see, however, that *all* the arithmetic operations on numbers of length N can in fact be done in $O(N \times \log N \times \log\log N)$ operations.

The trick is to recognize that multiplication is essentially a *convolution* (§13.1) of the digits of the multiplicand and multiplier, followed by some kind of carry operation. Consider, for example, two ways of writing the calculation 456×789:

$$\begin{array}{r} 456 \\ \times\ 789 \\ \hline 4104 \\ 3648 \\ 3192 \\ \hline 359784 \end{array} \qquad \begin{array}{rrrrrr} & & & 4 & 5 & 6 \\ & & \times & 7 & 8 & 9 \\ \hline & & & 36 & 45 & 54 \\ & & 32 & 40 & 48 & \\ & 28 & 35 & 42 & & \\ \hline & 28 & 67 & 118 & 93 & 54 \\ \hline 3 & 5 & 9 & 7 & 8 & 4 \end{array}$$

The tableau on the left shows the conventional method of multiplication, in which three separate short multiplications of the full multiplicand (by 9, 8, and 7) are added to obtain the final result. The tableau on the right shows a different method (sometimes taught for mental arithmetic), where the single-digit cross products are all computed (e.g. $8 \times 6 = 48$), then added in columns to obtain an incompletely carried result (here, the list $28, 67, 118, 93, 54$). The final step is a single pass from right to left, recording the single least-significant digit and carrying the higher digit or digits into the total to the left (e.g. $93 + 5 = 98$, record the 8, carry 9).

You can see immediately that the column sums in the right-hand method are components of the convolution of the digit strings, for example $118 = 4 \times 9 + 5 \times 8 + 6 \times 7$. In §13.1 we learned how to compute the convolution of two vectors by the fast Fourier transform (FFT): Each vector is FFT'd, the two complex transforms are multiplied, and the result is inverse-FFT'd. Since the transforms are done with floating arithmetic, we need sufficient precision so that the exact integer value of each component of the result is discernible in the presence of roundoff error. We should therefore allow a (conservative) few times $\log_2(\log_2 N)$ bits for roundoff in the FFT. A number of length N bytes in radix 256 can generate convolution components as large as the order of $(256)^2 N$, thus requiring $16 + \log_2 N$ bits of precision for exact storage. If `it` is the number of bits in the floating mantissa (cf. §20.1), we obtain the condition

$$16 + \log_2 N + \text{few} \times \log_2 \log_2 N < \texttt{it} \tag{20.6.3}$$

We see that single precision, say with $\texttt{it} = 24$, is inadequate for any interesting value of N, while double precision, say with $\texttt{it} = 53$, allows N to be greater than 10^6, corresponding to some millions of decimal digits. The following routine

therefore presumes double precision versions of `realft` (§12.3) and `four1` (§12.2), here called `drealft` and `dfour1`. (These routines are included on the *Numerical Recipes* diskettes.)

```
      SUBROUTINE mpmul(w,u,v,n,m)
      INTEGER m,n,NMAX
      CHARACTER*1 w(n+m),u(n),v(m)
      DOUBLE PRECISION RX
      PARAMETER (NMAX=8192,RX=256.D0)
C     USES drealft                DOUBLE PRECISION version of realft.
        Uses Fast Fourier Transform to multiply the unsigned radix 256 integers u(1:n) and
        v(1:m), yielding a product w(1:n+m).
      INTEGER j,mn,nn
      DOUBLE PRECISION cy,t,a(NMAX),b(NMAX)
      mn=max(m,n)
      nn=1                        Find the smallest useable power of two for the transform.
1     if(nn.lt.mn) then
          nn=nn+nn
      goto 1
      endif
      nn=nn+nn
      if(nn.gt.NMAX)pause 'NMAX too small in fftmul'
      do 11 j=1,n                 Move U to a double precision floating array.
          a(j)=ichar(u(j))
      enddo 11
      do 12 j=n+1,nn
          a(j)=0.D0
      enddo 12
      do 13 j=1,m                 Move V to a double precision floating array.
          b(j)=ichar(v(j))
      enddo 13
      do 14 j=m+1,nn
          b(j)=0.D0
      enddo 14                    Perform the convolution: First, the two Fourier transforms.
      call drealft(a,nn,1)
      call drealft(b,nn,1)
      b(1)=b(1)*a(1)              Then multiply the complex results (real and imaginary parts).
      b(2)=b(2)*a(2)
      do 15 j=3,nn,2
          t=b(j)
          b(j)=t*a(j)-b(j+1)*a(j+1)
          b(j+1)=t*a(j+1)+b(j+1)*a(j)
      enddo 15
      call drealft(b,nn,-1)       Then do the inverse Fourier transform.
      cy=0.                       Make a final pass to do all the carries.
      do 16 j=nn,1,-1
          t=b(j)/(nn/2)+cy+0.5D0  The 0.5 allows for roundoff error.
          b(j)=mod(t,RX)
          cy=int(t/RX)
      enddo 16
      if (cy.ge.RX) pause 'cannot happen in fftmul'
      w(1)=char(int(cy))          Copy answer to output.
      do 17 j=2,n+m
          w(j)=char(int(b(j-1)))
      enddo 17
      return
      END
```

With multiplication thus a "fast" operation, division is best performed by multiplying the dividend by the reciprocal of the divisor. The reciprocal of a value

V is calculated by iteration of Newton's rule,

$$U_{i+1} = U_i(2 - VU_i) \tag{20.6.4}$$

which results in the quadratic convergence of U_∞ to $1/V$, as you can easily prove. (Many supercomputers and RISC machines actually use this iteration to perform divisions.) We can now see where the operations count $N \log N \log\log N$, mentioned above, originates: $N \log N$ is in the Fourier transform, with the iteration to converge Newton's rule giving an additional factor of $\log\log N$.

```
      SUBROUTINE mpinv(u,v,n,m)
      INTEGER m,n,MF,NMAX
      CHARACTER*1 u(n),v(m)
      REAL BI
      PARAMETER (MF=4,BI=1./256.,NMAX=8192)
```

Character string `v(1:m)` is interpreted as a radix 256 number with the radix point after (nonzero) `v(1)`; `u(1:n)` is set to the most significant digits of its reciprocal, with the radix point after `u(1)`.

```
C     USES mpmov,mpmul,mpneg
      INTEGER i,j,mm
      REAL fu,fv
      CHARACTER*1 rr(2*NMAX+1),s(NMAX)
      if(max(n,m).gt.NMAX)pause 'NMAX too small in mpinv'
      mm=min(MF,m)
      fv=ichar(v(mm))
      do 11 j=mm-1,1,-1
        fv=fv*BI+ichar(v(j))
      enddo 11
      fu=1./fv
      do 12 j=1,n
        i=int(fu)
        u(j)=char(i)
        fu=256.*(fu-i)
      enddo 12
1     continue
        call mpmul(rr,u,v,n,m)
        call mpmov(s,rr(2),n)
        call mpneg(s,n)
        s(1)=char(ichar(s(1))-254)
        call mpmul(rr,s,u,n,n)
        call mpmov(u,rr(2),n)
        do 13 j=2,n-1
          if(ichar(s(j)).ne.0)goto 1
        enddo 13
      continue
      return
      END
```

Use ordinary floating arithmetic to get an initial approximation.

Iterate Newton's rule to convergence.

Construct $2 - UV$ in S.

Multiply SU into U.

If fractional part of S is not zero, it has not converged to 1.

Division now follows as a simple corollary, with only the necessity of calculating the reciprocal to sufficient accuracy to get an exact quotient and remainder.

```
      SUBROUTINE mpdiv(q,r,u,v,n,m)
      INTEGER m,n,NMAX,MACC
      CHARACTER*1 q(n-m+1),r(m),u(n),v(m)
      PARAMETER (NMAX=8192,MACC=6)
```

Divides unsigned radix 256 integers `u(1:n)` by `v(1:m)` (with `m` $\leq$ `n` required), yielding a quotient `q(1:n-m+1)` and a remainder `r(1:m)`.

```
C     USES mpinv,mpmov,mpmul,mpsad,mpsub
      INTEGER is
      CHARACTER*1 rr(2*NMAX),s(2*NMAX)
      if(n+MACC.gt.NMAX)pause 'NMAX too small in mpdiv'
```

```
      call mpinv(s,v,n+MACC,m)          Set S = 1/V.
      call mpmul(rr,s,u,n+MACC,n)       Set Q = SU.
      call mpsad(s,rr,n+n+MACC/2,1)
      call mpmov(q,s(3),n-m+1)
      call mpmul(rr,q,v,n-m+1,m)        Multiply and subtract to get the remainder.
      call mpsub(is,rr(2),u,rr(2),n)
      if (is.ne.0) pause 'MACC too small in mpdiv'
      call mpmov(r,rr(n-m+2),m)
      return
      END
```

Square roots are calculated by a Newton's rule much like division. If

$$U_{i+1} = \frac{1}{2}U_i(3 - VU_i^2) \tag{20.6.5}$$

then U_∞ converges quadratically to $1/\sqrt{V}$. A final multiplication by V gives $\sqrt{V}$.

```
      SUBROUTINE mpsqrt(w,u,v,n,m)
      INTEGER m,n,NMAX,MF
      CHARACTER*1 w(*),u(*),v(*)
      REAL BI
      PARAMETER (NMAX=2048,MF=3,BI=1./256.)
C     USES mplsh,mpmov,mpmul,mpneg,mpsdv
```

Character string `v(1:m)` is interpreted as a radix 256 number with the radix point after `v(1)`; `w(1:n)` is set to its square root (radix point after `w(1)`), and `u(1:n)` is set to the reciprocal thereof (radix point before `u(1)`). `w` and `u` need not be distinct, in which case they are set to the square root.

```
      INTEGER i,ir,j,mm
      REAL fu,fv
      CHARACTER*1 r(NMAX),s(NMAX)
      if(2*n+1.gt.NMAX)pause 'NMAX too small in mpsqrt'
      mm=min(m,MF)
      fv=ichar(v(mm))                   Use ordinary floating arithmetic to get an initial approx-
      do 11 j=mm-1,1,-1                     imation.
          fv=BI*fv+ichar(v(j))
      enddo 11
      fu=1./sqrt(fv)
      do 12 j=1,n
          i=int(fu)
          u(j)=char(i)
          fu=256.*(fu-i)
      enddo 12
1     continue                          Iterate Newton's rule to convergence.
          call mpmul(r,u,u,n,n)         Construct S = (3 - VU^2)/2.
          call mplsh(r,n)
          call mpmul(s,r,v,n,m)
          call mplsh(s,n)
          call mpneg(s,n)
          s(1)=char(ichar(s(1))-253)
          call mpsdv(s,s,n,2,ir)
          do 13 j=2,n-1                 If fractional part of S is not zero, it has not converged
              if(ichar(s(j)).ne.0)goto 2    to 1.
          enddo 13
              call mpmul(r,u,v,n,m)     Get square root from reciprocal and return.
              call mpmov(w,r(2),n)
              return
2         continue
          call mpmul(r,s,u,n,n)         Replace U by SU.
          call mpmov(u,r(2),n)
      goto 1
      END
```

We already mentioned that radix conversion to decimal is a merely cosmetic operation that should normally be omitted. The simplest way to convert a fraction to decimal is to multiply it repeatedly by 10, picking off (and subtracting) the resulting integer part. This, has an operations count of $O(N^2)$, however, since each liberated decimal digit takes an $O(N)$ operation. It *is* possible to do the radix conversion as a fast operation by a "divide and conquer" strategy, in which the fraction is (fast) multiplied by a large power of 10, enough to move about half the desired digits to the left of the radix point. The integer and fractional pieces are now processed independently, each further subdivided. If our goal were a few billion digits of π, instead of a few thousand, we would need to implement this scheme. For present purposes, the following lazy routine is adequate:

```
      SUBROUTINE mp2dfr(a,s,n,m)
      INTEGER m,n,IAZ
      CHARACTER*1 a(*),s(*)
      PARAMETER (IAZ=48)
C     USES mplsh,mpsmu
        Converts a radix 256 fraction a(1:n) (radix point before a(1)) to a decimal fraction
        represented as an ascii string s(1:m), where m is a returned value. The input array a(1:n)
        is destroyed. NOTE: For simplicity, this routine implements a slow (∝ N^2) algorithm. Fast
        (∝ N ln N), more complicated, radix conversion algorithms do exist.
      INTEGER j
          m=2.408*n
          do 11 j=1,m
              call mpsmu(a,a,n,10)
              s(j)=char(ichar(a(1))+IAZ)
              call mplsh(a,n)
          enddo 11
      return
      END
```

Finally, then, we arrive at a routine implementing equations (20.6.1) and (20.6.2):

```
      SUBROUTINE mppi(n)
      INTEGER n,IAOFF,NMAX
      PARAMETER (IAOFF=48,NMAX=8192)
C     USES mpinit,mp2dfr,mpadd,mpinv,mplsh,mpmov,mpmul,mpsdv,mpsqrt
        Demonstrate multiple precision routines by calculating and printing the first n bytes of π.
      INTEGER ir,j,m
      CHARACTER*1 x(NMAX),y(NMAX),sx(NMAX),sxi(NMAX),t(NMAX),s(3*NMAX),
     *     pi(NMAX)
      call mpinit
      t(1)=char(2)                          Set T = 2.
      do 11 j=2,n
          t(j)=char(0)
      enddo 11
      call mpsqrt(x,x,t,n,n)                Set X_0 = √2.
      call mpadd(pi,t,x,n)                  Set π_0 = 2 + √2.
      call mplsh(pi,n)
      call mpsqrt(sx,sxi,x,n,n)             Set Y_0 = 2^(1/4).
      call mpmov(y,sx,n)
1     continue
          call mpadd(x,sx,sxi,n)            Set X_{i+1} = (X_i^{1/2} + X_i^{-1/2})/2.
          call mpsdv(x,x(2),n,2,ir)
          call mpsqrt(sx,sxi,x,n,n)         Form the temporary T = Y_i X_{i+1}^{1/2} + X_{i+1}^{-1/2}.
          call mpmul(t,y,sx,n,n)
          call mpadd(t(2),t(2),sxi,n)
```

3.14159265358979323846264338327950288419716939937510582097494459230781640628
62089986280348253421170679821480865132823066470938446095505822317253594081
28481117450284102701938521105559644622948954930381964428810975665933446128 4
75648233786783165271201909145648566923460348610454326648213393607260249141 2
73724587006606315588174881520920962829254091715364367892590360011330530548 8
20466521384146951941511609433057270365759591953092186117381932611793105118 5
48074462379962749567351885752724891227938183011949129833673362440656643086 0
21394946395224737190702179860943702770539217176293176752384674818467669405 1
32000568127145263560827785771342757789609173637178721468440901224953430146 5
49585371050792279689258923542019956112129021960864034418159813629774771309 9
60518707211349999998372978049951059731732816096318595024459455346908302642 5
22308253344685035261931188171010003137838752886587533208381420617177669147 3
03598253490428755468731159562863882353787593751957781857780532171226806613 0
01927876611195909216420198938095257201065485863278865936153381827968230301 9
52035301852968995773622599413891249721775283479131515574857242454150695950 8
29533116861727855889075098381754637464939319255060400927701671139009848824 0
12858361603563707660104710181942955596198946767837449448255379774726847104 0
47534646208046684259069491293313677028989152104752162056966024058038150193 5
11253382430035587640247496473263914199272604269922796782354781636009341721 6
41219924586315030286182974555706749838505494588586926995690927210797509302 9
55321165344987202755960236480665499119881834797753566369807426542578625518
18417574672890977772793800081647060016145249192173217214772350141441973568 5
48161361157352552133475741849468438523323907394143334547762416825189835694
85562099219222184272550254256887671790494601653466804988627232791786085784 3
83827967976681454100953883786360950680064225125205117392984896084128488626 9
45604241965285022210661186306744278622039194945047123713786960956364371917 2
87467764657573962413890865832645995813390478027590099465764078951269468398 3
52595709825822620522489407726719478268482601476990902640136394437455305068 2
03496252451749399651431429809190659250937221696461515709858387410597885959 7
72975498930161753928468138268683868942774155991855925245953959431049972524 6
80845987273644695848653836736222626099124608051243884390451244136549762780 7
97715691435997700129616089441694868555848406353422072225828488648158456028 5

Figure 20.6.1. The first 2398 decimal digits of π, computed by the routines in this section.

```
    x(1)=char(ichar(x(1))+1)          Increment X_{i+1} and Y_i by 1.
    y(1)=char(ichar(y(1))+1)
    call mpinv(s,y,n,n)               Set Y_{i+1} = T/(Y_i + 1).
    call mpmul(y,t(3),s,n,n)
    call mplsh(y,n)
    call mpmul(t,x,s,n,n)             Form temporary T = (X_{i+1} + 1)/(Y_i + 1).
    continue                          If T = 1 then we have converged.
        m=mod(255+ichar(t(2)),256)
        do 12 j=3,n
            if(ichar(t(j)).ne.m)goto 2
        enddo 12
        if (abs(ichar(t(n+1))-m).gt.1)goto 2
        write (*,*) 'pi='
        s(1)=char(ichar(pi(1))+IAOFF)
        s(2)='.'
        call mp2dfr(pi(2),s(3),n-1,m)
          Convert to decimal for printing. NOTE: The conversion routine, for this demonstration only, is a slow (∝ N^2) algorithm. Fast (∝ N ln N), more complicated, radix conversion algorithms do exist.
        write (*,'(1x,64a1)') (s(j),j=1,m+1)
        return
2       continue
        call mpmul(s,pi,t(2),n,n)     Set π_{i+1} = T π_i.
        call mpmov(pi,s(2),n)
    goto 1
    END
```

Figure 20.6.1 gives the result, computed with $\mathtt{n} = 1000$. As an exercise, you might enjoy checking the first hundred digits of the figure against the first 12 terms of Ramanujan's celebrated identity [3]

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!\,(1103 + 26390n)}{(n!\,396^n)^4} \tag{20.6.6}$$

using the above routines. You might also use the routines to verify that the number $2^{512} + 1$ is not a prime, but has factors 2,424,833 and 7,455,602,825,647,884,208,337,395,736,200,454,918,783,366,342,657 (which are in fact prime; the remaining prime factor being about 7.416×10^{98}) [4].

CITED REFERENCES AND FURTHER READING:

Borwein, J.M., and Borwein, P.B. 1987, *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity* (New York: Wiley). [1]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.3. [2]

Ramanujan, S. 1927, *Collected Papers of Srinivasa Ramanujan*, G.H. Hardy, P.V. Seshu Aiyar, and B.M. Wilson, eds. (Cambridge, U.K.: Cambridge University Press), pp. 23–39. [3]

Kolata, G. 1990, June 20, *The New York Times*. [4]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).

References

The references collected here are those of general usefulness, usually cited in more than one section of this book. More specialized sources, usually cited in a single section, are not repeated here.

We first list a small number of books that form the nucleus of a recommended personal reference collection on numerical methods, numerical analysis, and closely related subjects. These are the books that we like to have within easy reach.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York)

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America)

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press)

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag)

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall)

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press)

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall)

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley)

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press)

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall)

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill)

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley)

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag)

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag)

We next list the larger collection of books, which, in our view, should be included in any serious research library on computing, numerical methods, or analysis.

Bevington, P.R. 1969, *Data Reduction and Error Analysis for the Physical Sciences* (New York: McGraw-Hill)

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley)

Bowers, R.L., and Wilson, J.R. 1991, *Numerical Modeling in Applied Physics and Astrophysics* (Boston: Jones & Bartlett)

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall)

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall)

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley)

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press)

Canuto, C., Hussaini, M.Y., Quarteroni, A., and Zang, T.A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag)

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley)

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press)

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press)

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders)

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press)

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag)

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.)

Downie, N.M., and Heath, R.W. 1965, *Basic Statistical Methods*, 2nd ed. (New York: Harper & Row)

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.)

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press)

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall)

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall)

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall)

Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill)

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall)

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library)

Gottlieb, D. and Orszag, S.A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphia: S.I.A.M.)

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag)

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists*; reprinted 1986 (New York: Dover)

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley)

Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press)

Hildebrand, F.B. 1974, *Introduction to Numerical Analysis*, 2nd ed.; reprinted 1987 (New York: Dover)

Hoel, P.G. 1971, *Introduction to Mathematical Statistics*, 4th ed. (New York: Wiley)

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press)

Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill)

Huber, P.J. 1981, *Robust Statistics* (New York: Wiley)

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley)

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press)

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley)

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall)

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell)

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Koonin, S.E., and Meredith, D.C. 1990, *Computational Physics, Fortran Version* (Redwood City, CA: Addison-Wesley)

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press)

Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover)

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience)

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall)

Lehmann, E.L. 1975, *Nonparametrics: Statistical Methods Based on Ranks* (San Francisco: Holden-Day)

Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press)

Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea)

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press)

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley)

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press)

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley)

Norusis, M.J. 1982, *SPSS Introductory Guide: Basic Statistics and Operations*; and 1985, *SPSS-X Advanced Statistics Guide* (New York: McGraw-Hill)

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag)

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press)

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press)

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press)

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill)

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience)

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa)

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall)

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag)

Stuart, A., and Ord, J.K. 1987, *Kendall's Advanced Theory of Statistics*, 5th ed. (London: Griffin and Co.) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*]

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press)

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley)

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press)

Young, D.M., and Gregory, R.T. 1973, *A Survey of Numerical Mathematics*, 2 vols.; reprinted 1988 (New York: Dover)

Index of Programs and Dependencies

The following table lists, in alphabetical order, all the routines in *Numerical Recipes*. When a routine requires subsidiary routines, either from this book or else user-supplied, the full dependency tree is shown: A routine calls directly all routines to which it is connected by a solid line in the column immediately to its right; it calls indirectly the connected routines in all columns to its right. Typographical conventions: Routines from this book are in typewriter font (e.g., `eulsum`, *`gammln`*). The smaller, slanted font is used for the second and subsequent occurences of a routine in a single dependency tree. (When you are getting routines from the *Numerical Recipes* diskettes, or their archive files, you need only specify names in the larger, upright font.) User-supplied routines are indicated by the use of text font and square brackets, e.g., [funcv]. Consult the text for individual specifications of these routines. The right-hand side of the table lists section and page numbers for each program.

`addint` — `interp` . . . . . §19.6 (p. 871)

`airy` — `bessik` — `beschb` — `chebev`
`bessjy` —

`amebsa` — `ran1` . . . . . §10.9 (p. 445)
`amotsa` — [funk]
`ran1`
[funk]

`amoeba` — `amotry` — [funk] . . . . . §10.4 (p. 404)
[funk]

`amotry` — [funk] . . . . . §10.4 (p. 405)

`amotsa` — [funk] . . . . . §10.9 (p. 446)
`ran1`

`anneal` — `ran3` . . . . . §10.9 (p. 439)
`irbit1`
`trncst`
`metrop` — *`ran3`*
`trnspt`
`revcst`
`revers`

`anorm2` . . . . . §19.6 (p. 879)

`arcmak` . . . . . §20.5 (p. 904)

`arcode` — `arcsum` . . . . . §20.5 (p. 904)

`arcsum` . . . . . §20.5 (p. 906)

`avevar` . . . . . §14.2 (p. 611)

`badluk` — `julday` . . . . . §1.1 (p. 14)
`flmoon`

`balanc` . . . . . §11.5 (p. 477)

banbks . . . . . §2.4 (p. 46)
bandec . . . . . §2.4 (p. 45)
banmul . . . . . §2.4 (p. 44)
bcucof . . . . . §3.6 (p. 119)
bcuint — bcucof . . . . . §3.6 (p. 120)
beschb — chebev . . . . . §6.7 (p. 239)
bessi — bessi0 . . . . . §6.6 (p. 233)
bessi0 . . . . . §6.6 (p. 230)
bessi1 . . . . . §6.6 (p. 231)
bessik — beschb — chebev . . . . . §6.7 (p. 241)
bessj — bessj0 . . . . . §6.5 (p. 228)
  └ bessj1
bessj0 . . . . . §6.5 (p. 225)
bessj1 . . . . . §6.5 (p. 226)
bessjy — beschb — chebev . . . . . §6.7 (p. 236)
bessk — bessk0 — bessi0 . . . . . §6.6 (p. 232)
  └ bessk1 — bessi1
bessk0 — bessi0 . . . . . §6.6 (p. 231)
bessk1 — bessi1 . . . . . §6.6 (p. 232)
bessy — bessy1 — bessj1 . . . . . §6.5 (p. 227)
  └ bessy0 — bessj0
bessy0 — bessj0 . . . . . §6.5 (p. 226)
bessy1 — bessj1 . . . . . §6.5 (p. 227)
beta — gammln . . . . . §6.1 (p. 209)
betacf . . . . . §6.4 (p. 221)
betai — gammln . . . . . §6.4 (p. 220)
  └ betacf
bico — factln — gammln . . . . . §6.1 (p. 208)
bksub . . . . . §17.3 (p. 761)
bnldev — ran1 . . . . . §7.3 (p. 285)
  └ gammln
brent — [func] . . . . . §10.2 (p. 397)
broydn — fmin — . . . . . §9.7 (p. 383)
  ├ fdjac — [funcv]
  ├ qrdcmp
  ├ qrupdt — rotate
  ├ rsolv
  └ lnsrch — *fmin* — [funcv]
bsstep — mmid — [derivs] . . . . . §16.4 (p. 722)
  └ pzextr
caldat . . . . . §1.1 (p. 16)

chder . . . . . §5.9 (p. 189)

chebev . . . . . §5.8 (p. 187)

chebft — [func] . . . . . §5.8 (p. 186)

chebpc . . . . . §5.10 (p. 191)

chint . . . . . §5.9 (p. 189)

chixy . . . . . §15.3 (p. 663)

choldc . . . . . §2.9 (p. 90)

cholsl . . . . . §2.9 (p. 90)

chsone — gammq — gser — gammln
gcf —

chstwo — gammq — gser — gammln
gcf —

cisi . . . . . §6.9 (p. 251)

cntab1 — gammq — gser — gammln
gcf —

cntab2 . . . . . §14.4 (p. 629)

convlv — twofft — four1
realft —

copy . . . . . §19.6 (p. 873)

correl — twofft — four1
realft —

cosft1 — realft — four1 . . . . . §12.3 (p. 512)

cosft2 — realft — four1 . . . . . §12.3 (p. 514)

covsrt . . . . . §15.4 (p. 669)

crank . . . . . §14.6 (p. 636)

cyclic — tridag . . . . . §2.7 (p. 68)

daub4 . . . . . §13.10 (p. 588)

dawson . . . . . §6.10 (p. 253)

dbrent — [func] . . . . . §10.3 (p. 400)
[dfunc]

ddpoly . . . . . §5.3 (p. 168)

decchk . . . . . §20.3 (p. 895)

df1dim — [dfunc] . . . . . §10.6 (p. 417)

dfour1 . . . . . DOUBLE PRECISION version of four1, *q.v.*

dfpmin — [func] . . . . . §10.7 (p. 421)
[dfunc]
lnsrch — [func]

dfridr — [func] . . . . . §5.7 (p. 182)

dftcor . . . . . §13.9 (p. 580)

dftint ─┬─ [func] . . . . . §13.9 (p. 581)
        ├─ realft ── four1
        ├─ polint
        └─ dftcor

difeq . . . . . §17.4 (p. 769)

dpythag . . . . . DOUBLE PRECISION version of pythag, *q.v.*

drealft . . . . . DOUBLE PRECISION version of realft, *q.v.*

dsprsax . . . . . DOUBLE PRECISION version of sprsax, *q.v.*

dsprstx . . . . . DOUBLE PRECISION version of sprstx, *q.v.*

dsvbksb . . . . . DOUBLE PRECISION version of svbksb, *q.v.*

dsvdcmp . . . . . DOUBLE PRECISION version of svdcmp, *q.v.*

eclass . . . . . §8.6 (p. 338)

eclazz ── [equiv] . . . . . §8.6 (p. 339)

ei . . . . . §6.3 (p. 218)

eigsrt . . . . . §11.1 (p. 462)

elle ─┬─ rf . . . . . §6.11 (p. 261)
      └─ rd

ellf ── rf . . . . . §6.11 (p. 260)

ellpi ─┬─ rf . . . . . §6.11 (p. 261)
       └─ rj ─┬─ rc
              └─ *rf*

elmhes . . . . . §11.5 (p. 479)

erf ── gammp ─┬─ gser ─┐ . . . . . §6.2 (p. 213)
              └─ gcf ──┴─ gammln

erfc ─┬─ gammp ─┬─ gser ─┐ . . . . . §6.2 (p. 214)
      │   └─ gcf ──┴─ gammln
      └─ gammq ─┬─ *gser* ─┐
                └─ *gcf* ──┴─ *gammln*

erfcc . . . . . §6.2 (p. 214)

eulsum . . . . . §5.1 (p. 161)

evlmem . . . . . §13.7 (p. 567)

expdev ── ran1 . . . . . §7.2 (p. 278)

expint . . . . . §6.3 (p. 217)

f1dim ── [func] . . . . . §10.5 (p. 413)

factln ── gammln . . . . . §6.1 (p. 208)

factrl ── gammln . . . . . §6.1 (p. 207)

fasper ─┬─ avevar . . . . . §13.8 (p. 575)
        ├─ spread
        └─ realft ── four1

fdjac ── [funcv] . . . . . §9.7 (p. 381)

fgauss . . . . . §15.5 (p. 683)

fill0 . . . . . §19.6 (p. 873)

fit — gammq — gser — gcf — gammln . . . . . §15.2 (p. 659)

fitexy — avevar . . . . . §15.3 (p. 662)
- fit — gammq — gser — gcf — gammln
- chixy
- mnbrak
- brent
- *gammq* — *gser* — *gcf* — *gammln*
- zbrent — *chixy*

fixrts — zroots — laguer . . . . . §13.6 (p. 562)

fleg . . . . . §15.4 (p. 674)

flmoon . . . . . §1.0 (p. 1)

fmin — [funcv] . . . . . §9.7 (p. 381)

four1 . . . . . §12.2 (p. 501)

fourew . . . . . §12.6 (p. 528)

fourfs — fourew . . . . . §12.6 (p. 525)

fourn . . . . . §12.4 (p. 518)

fpoly . . . . . §15.4 (p. 674)

fred2 — gauleg . . . . . §18.1 (p. 784)
- [ak]
- [g]
- ludcmp
- lubksb

fredex — quadmx — wwghts — kermom . . . . . §18.3 (p. 793)
- ludcmp
- lubksb

fredin — [ak] . . . . . §18.1 (p. 784)
- [g]

frenel . . . . . §6.9 (p. 249)

frprmn — [func] . . . . . §10.6 (p. 416)
- [dfunc]
- linmin — mnbrak — brent — f1dim — [func]

ftest — avevar . . . . . §14.2 (p. 613)
- betai — gammln
- betacf

gamdev — ran1 . . . . . §7.3 (p. 283)

gammln . . . . . §6.1 (p. 207)

gammp — gser — gcf — gammln . . . . . §6.2 (p. 211)

gammq — gser — gammln; gcf — gammln . . . §6.2 (p. 211)
gasdev — ran1 . . . §7.2 (p. 280)
gaucof — tqli — pythag; eigsrt . . . §4.5 (p. 151)
gauher . . . §4.5 (p. 147)
gaujac — gammln . . . §4.5 (p. 148)
gaulag — gammln . . . §4.5 (p. 146)
gauleg . . . §4.5 (p. 145)
gaussj . . . §2.1 (p. 30)
gcf — gammln . . . §6.2 (p. 212)
golden — [func] . . . §10.1 (p. 394)
gser — gammln . . . §6.2 (p. 212)
hpsel — sort . . . §8.5 (p. 336)
hpsort . . . §8.3 (p. 329)
hqr . . . §11.6 (p. 484)
hufapp . . . §20.4 (p. 899)
hufdec . . . §20.4 (p. 900)
hufenc . . . §20.4 (p. 900)
hufmak — hufapp . . . §20.4 (p. 898)
hunt . . . §3.4 (p. 112)
hypdrv . . . §6.12 (p. 265)
hypgeo — hypser . . . §6.12 (p. 264)
hypgeo — odeint — bsstep — mmid; pzextr; hypdrv
hypser . . . §6.12 (p. 264)
icrc — icrc1 . . . §20.3 (p. 893)
icrc1 . . . §20.3 (p. 892)
igray . . . §20.2 (p. 888)
iindexx . . . INTEGER version of indexx, *q.v.*
indexx . . . §8.4 (p. 330)
interp . . . §19.6 (p. 871)
irbit1 . . . §7.4 (p. 288)
irbit2 . . . §7.4 (p. 290)
jacobi . . . §11.1 (p. 460)
jacobn . . . §16.6 (p. 734)
julday . . . §1.1 (p. 13)
kendl1 — erfcc . . . §14.6 (p. 638)
kendl2 — erfcc . . . §14.6 (p. 639)
kermom . . . §18.3 (p. 792)

ks2d1s
- quadct . . . §14.7 (p. 642)
- quadvl
- pearsn — betai
  - gammln
  - betacf
- probks

ks2d2s
- quadct . . . §14.7 (p. 643)
- pearsn — betai
  - gammln
  - betacf
- probks

ksone
- sort . . . §14.3 (p. 619)
- [func]
- probks

kstwo
- sort . . . §14.3 (p. 619)
- probks

laguer . . . §9.5 (p. 366)

lfit
- [funcs] . . . §15.4 (p. 668)
- gaussj
- covsrt

linbcg
- atimes . . . §2.7 (p. 79)
- snrm
- asolve

linmin
- mnbrak . . . §10.5 (p. 412)
- brent — f1dim — [func]

lnsrch — [func] . . . §9.7 (p. 378)

locate . . . §3.4 (p. 111)

lop . . . §19.6 (p. 879)

lubksb . . . §2.3 (p. 39)

ludcmp . . . §2.3 (p. 38)

machar . . . §20.1 (p. 884)

maloc . . . §19.6 (p. 873)

matadd . . . §19.6 (p. 879)

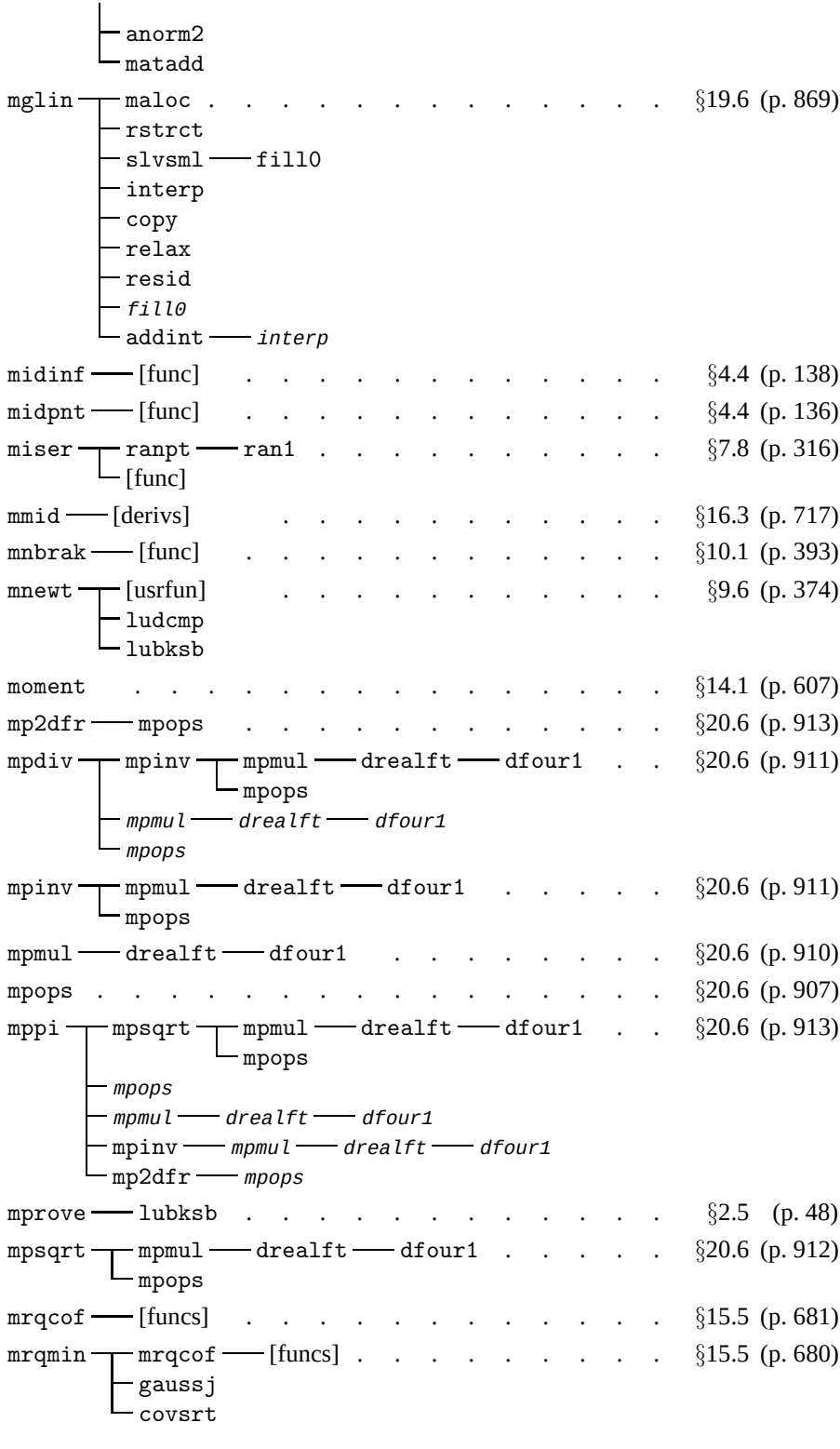
matsub . . . §19.6 (p. 879)

medfit — rofunc — select . . . §15.7 (p. 699)

memcof . . . §13.6 (p. 561)

metrop — ran3 . . . §10.9 (p. 443)

mgfas
- maloc . . . §19.6 (p. 877)
- rstrct
- slvsm2 — fill0
- interp
- copy
- relax2
- lop
- matsub


anorm2
matadd
mglin maloc §19.6 (p. 869)
rstrct
slvsml fill0
interp
copy
relax
resid
fill0
addint interp
midinf [func] §4.4 (p. 138)
midpnt [func] §4.4 (p. 136)
miser ranpt ran1 §7.8 (p. 316)
[func]
mmid [derivs] §16.3 (p. 717)
mnbrak [func] §10.1 (p. 393)
mnewt [usrfun] §9.6 (p. 374)
ludcmp
lubksb
moment §14.1 (p. 607)
mp2dfr mpops §20.6 (p. 913)
mpdiv mpinv mpmul drealft dfour1 §20.6 (p. 911)
mpops
mpmul drealft dfour1
mpops
mpinv mpmul drealft dfour1 §20.6 (p. 911)
mpops
mpmul drealft dfour1 §20.6 (p. 910)
mpops §20.6 (p. 907)
mppi mpsqrt mpmul drealft dfour1 §20.6 (p. 913)
mpops
mpops
mpmul drealft dfour1
mpinv mpmul drealft dfour1
mp2dfr mpops
mprove lubksb §2.5 (p. 48)
mpsqrt mpmul drealft dfour1 §20.6 (p. 912)
mpops
mrqcof [funcs] §15.5 (p. 681)
mrqmin mrqcof [funcs] §15.5 (p. 680)
gaussj
covsrt

newt — fmin, fdjac — [funcv], ludcmp, lubksb, lnsrch — *fmin* — [funcv] . . . §9.7 (p. 379)

odeint — [derivs], rkqs — [derivs], rkck — [derivs] . . . §16.2 (p. 714)

orthog . . . §4.5 (p. 153)

pade — ludcmp, lubksb, mprove — *lubksb* . . . §5.12 (p. 196)

pccheb . . . §5.11 (p. 193)

pcshft . . . §5.10 (p. 192)

pearsn — betai — gammln, betacf . . . §14.5 (p. 632)

period — avevar . . . §13.8 (p. 572)

piksr2 . . . §8.1 (p. 322)

piksrt . . . §8.1 (p. 321)

pinvs . . . §17.3 (p. 762)

plgndr . . . §6.8 (p. 247)

poidev — ran1, gammln . . . §7.3 (p. 284)

polcoe . . . §3.5 (p. 114)

polcof — polint . . . §3.5 (p. 115)

poldiv . . . §5.3 (p. 169)

polin2 — polint . . . §3.6 (p. 118)

polint . . . §3.1 (p. 103)

powell — [func], linmin — mnbrak, brent — f1dim — [func] . . . §10.5 (p. 411)

predic . . . §13.6 (p. 562)

probks . . . §14.3 (p. 620)

psdes . . . §7.5 (p. 293)

pwt . . . §13.10 (p. 589)

pwtset . . . §13.10 (p. 589)

pythag . . . §2.6 (p. 62)

pzextr . . . §16.4 (p. 724)

qgaus — [func] . . . §4.5 (p. 141)

qrdcmp . . . §2.10 (p. 92)

qromb ─┬─ trapzd ── [func] . . . . . §4.3 (p. 134)
       └─ polint

qromo ─┬─ midpnt ── [func] . . . . . §4.4 (p. 137)
       └─ polint

qroot ── poldiv . . . . . §9.5 (p. 371)

qrsolv ── rsolv . . . . . §2.10 (p. 93)

qrupdt ── rotate . . . . . §2.10 (p. 94)

qsimp ── trapzd ── [func] . . . . . §4.2 (p. 133)

qtrap ── trapzd ── [func] . . . . . §4.2 (p. 131)

quad3d ── qgaus ─┬─ [func] . . . . . §4.6 (p. 157)
                 ├─ [y1]
                 ├─ [y2]
                 ├─ [z1]
                 └─ [z2]

quadct . . . . . §14.7 (p. 642)

quadmx ── wwghts ── kermom . . . . . §18.3 (p. 793)

quadvl . . . . . §14.7 (p. 643)

ran0 . . . . . §7.1 (p. 270)

ran1 . . . . . §7.1 (p. 271)

ran2 . . . . . §7.1 (p. 272)

ran3 . . . . . §7.1 (p. 273)

ran4 ── psdes . . . . . §7.5 (p. 294)

rank . . . . . §8.4 (p. 333)

ranpt ── ran1 . . . . . §7.8 (p. 318)

ratint . . . . . §3.2 (p. 106)

ratlsq ─┬─ [fn] . . . . . §5.13 (p. 200)
        ├─ dsvdcmp ── dpythag
        ├─ dsvbksb
        └─ ratval

ratval . . . . . §5.3 (p. 170)

rc . . . . . §6.11 (p. 259)

rd . . . . . §6.11 (p. 257)

realft ── four1 . . . . . §12.3 (p. 507)

rebin . . . . . §7.8 (p. 314)

red . . . . . §17.3 (p. 763)

relax . . . . . §19.6 (p. 872)

relax2 . . . . . §19.6 (p. 878)

resid . . . . . §19.6 (p. 872)

revcst . . . . . §10.9 (p. 441)

revers . . . . . §10.9 (p. 442)

rf . . . . . §6.11 (p. 257)

rj ┬ rc . . . . . §6.11 (p. 258)
   └ rf

rk4 ── [derivs] . . . . . §16.1 (p. 706)

rkck ── [derivs] . . . . . §16.2 (p. 713)

rkdumb ┬ [derivs] . . . . . §16.1 (p. 707)
       └ rk4 ── [derivs]

rkqs ── rkck ── [derivs] . . . . . §16.2 (p. 712)

rlft3 ── fourn . . . . . §12.5 (p. 522)

rofunc ── select . . . . . §15.7 (p. 700)

rotate . . . . . §2.10 (p. 95)

rsolv . . . . . §2.10 (p. 93)

rstrct . . . . . §19.6 (p. 870)

rtbis ── [func] . . . . . §9.1 (p. 347)

rtflsp ── [func] . . . . . §9.2 (p. 349)

rtnewt ── [funcd] . . . . . §9.4 (p. 358)

rtsafe ── [funcd] . . . . . §9.4 (p. 359)

rtsec ── [func] . . . . . §9.2 (p. 350)

rzextr . . . . . §16.4 (p. 725)

savgol ┬ ludcmp . . . . . §14.8 (p. 646)
       └ lubksb

scrsho ── [func] . . . . . §9.0 (p. 342)

select . . . . . §8.5 (p. 334)

selip ── shell . . . . . §8.5 (p. 335)

sfroid ┬ plgndr . . . . . §17.4 (p. 768)
       └ solvde ┬ difeq
                ├ pinvs
                ├ red
                └ bksub

shell . . . . . §8.1 (p. 323)

shoot ┬ [load] . . . . . §17.1 (p. 750)
      ├ odeint ┬ [derivs]
      │   └ rkqs ── rkck ── [derivs]
      └ [score]

shootf ┬ [load1] . . . . . §17.2 (p. 752)
       ├ odeint ┬ [derivs]
       │   └ rkqs ── rkck ── [derivs]
       ├ [score]
       └ [load2]

simp1 . . . . . §10.8 (p. 434)

simp2 . . . . . §10.8 (p. 434)

simp3 . . . . . §10.8 (p. 435)

simplx
- simp1 . . . . . §10.8 (p. 432)
- simp2
- simp3

simpr
- ludcmp . . . . . §16.6 (p. 736)
- lubksb
- [derivs]

sinft — realft — four1 . . . . . §12.3 (p. 511)

slvsm2 — fill0 . . . . . §19.6 (p. 878)

slvsml — fill0 . . . . . §19.6 (p. 872)

sncndn . . . . . §6.11 (p. 262)

snrm . . . . . §2.7 (p. 81)

sobseq . . . . . §7.7 (p. 302)

solvde
- difeq . . . . . §17.3 (p. 760)
- pinvs
- red
- bksub

sor . . . . . §19.5 (p. 860)

sort . . . . . §8.2 (p. 324)

sort2 . . . . . §8.2 (p. 326)

sort3 — indexx . . . . . §8.4 (p. 332)

spctrm — four1 . . . . . §13.4 (p. 550)

spear
- sort2 . . . . . §14.6 (p. 635)
- crank
- erfcc
- betai
  - gammln
  - betacf

sphbes — bessjy — beschb — chebev . . . . . §6.7 (p. 245)

sphfpt — newt
- fdjac — shootf *(q.v.)* . . . . §17.4 (p. 772)
- lnsrch
- fmin — *shootf (q.v.)*
- ludcmp
- lubksb

sphoot — newt
- fdjac — shoot *(q.v.)* . . . . §17.4 (p. 771)
- lnsrch
- fmin — *shoot (q.v.)*
- ludcmp
- lubksb

splie2 — spline . . . . . §3.6 (p. 121)

splin2
- splint . . . . . §3.6 (p. 121)
- spline

spline . . . . . §3.3 (p. 109)

splint . . . . . §3.3 (p. 110)

spread . . . . . §13.8 (p. 576)

sprsax . . . . . §2.7 (p. 72)

sprsin . . . . . §2.7 (p. 72)

sprspm . . . . . §2.7 (p. 75)

sprstm . . . . . §2.7 (p. 76)

sprstp
- iindexx . . . . . §2.7 (p. 73)

sprstx . . . . . §2.7 (p. 73)

stifbs
- jacobn . . . . . §16.6 (p. 737)
- simpr
  - ludcmp
  - lubksb
  - [derivs]
- pzextr

stiff
- jacobn . . . . . §16.6 (p. 732)
- ludcmp
- lubksb
- [derivs]

stoerm
- [derivs] . . . . . §16.5 (p. 726)

svbksb . . . . . §2.6 (p. 56)

svdcmp
- pythag . . . . . §2.6 (p. 59)

svdfit
- [funcs] . . . . . §15.4 (p. 672)
- svdcmp
  - pythag
- svbksb

svdvar . . . . . §15.4 (p. 673)

toeplz . . . . . §2.8 (p. 88)

tptest
- avevar . . . . . §14.2 (p. 612)
- betai
  - gammln
  - betacf

tqli
- pythag . . . . . §11.3 (p. 473)

trapzd
- [func] . . . . . §4.2 (p. 131)

tred2 . . . . . §11.2 (p. 467)

tridag . . . . . §2.4 (p. 43)

trncst . . . . . §10.9 (p. 442)

trnspt . . . . . §10.9 (p. 442)

ttest
- avevar . . . . . §14.2 (p. 610)
- betai
  - gammln
  - betacf

tutest
- avevar . . . . . §14.2 (p. 611)
- betai
  - gammln
  - betacf

twofft
- four1 . . . . . §12.3 (p. 505)

vander . . . . . §2.8 (p. 84)

vegas ┬ rebin . . . . . §7.8 (p. 311)
     ├ ran2
     └ [fxn]
voltra ┬ [g] . . . . . §18.2 (p. 787)
     ├ [ak]
     ├ ludcmp
     └ lubksb
wt1 ── daub4 . . . . . §13.10 (p. 587)
wtn ── daub4 . . . . . §13.10 (p. 595)
wwghts ── kermom . . . . . §18.3 (p. 791)
zbrac ── [func] . . . . . §9.1 (p. 345)
zbrak ── [func] . . . . . §9.1 (p. 345)
zbrent ── [func] . . . . . §9.3 (p. 354)
zrhqr ┬ balanc . . . . . §9.5 (p. 368)
     └ hqr
zriddr ── [func] . . . . . §9.2 (p. 351)
zroots ── laguer . . . . . §9.5 (p. 367)

General Index to Volumes 1 and 2

In this index, page numbers 1 through 934 refer to Volume 1, *Numerical Recipes in Fortran 77*, while page numbers 935 through 1446 refer to Volume 2, *Numerical Recipes in Fortran 90*. Front matter in Volume 1 is indicated by page numbers in the range 1/i through 1/xxxi, while front matter in Volume 2 is indicated 2/i through 2/xx.

Abstract data types 2/xiii, 1030
Accelerated convergence of series 160ff., 1070
Accuracy 19f.
  achievable in minimization 392, 397, 404
  achievable in root finding 346f.
  contrasted with fidelity 832, 840
  CPU different from memory 181
  vs. stability 704, 729, 830, 844
Accuracy parameters 1362f.
Acknowledgments 1/xvi, 2/ix
Ada 2/x
Adams-Bashford-Moulton method 741
Adams' stopping criterion 366
Adaptive integration 123, 135, 703, 708ff., 720, 726, 731f., 737, 742ff., 788, 1298ff., 1303, 1308f.
  Monte Carlo 306ff., 1161ff.
Addition, multiple precision 907, 1353
Addition theorem, elliptic integrals 255
ADI (alternating direction implicit) method 847, 861f., 906
Adjoint operator 867
Adobe Illustrator 1/xvi, 2/xx
Advective equation 826
AGM (arithmetic geometric mean) 906
Airy function 204, 234, 243f.
  routine for 244f., 1121
Aitken's delta squared process 160
Aitken's interpolation algorithm 102
Algol 2/x, 2/xiv
Algorithms, non-numerical 881ff., 1343ff.
Aliasing 495, 569
  see also Fourier transform
all() intrinsic function 945, 948
All-poles model 566
  see also Maximum entropy method (MEM)
All-zeros model 566
  see also Periodogram
Allocatable array 938, 941, 952ff., 1197, 1212, 1266, 1293, 1306, 1336
allocate statement 938f., 941, 953f., 1197, 1266, 1293, 1306, 1336
allocated() intrinsic function 938, 952ff., 1197, 1266, 1293
Allocation status 938, 952ff., 961, 1197, 1266, 1293
Alpha AXP 2/xix
Alternating-direction implicit method (ADI) 847, 861f., 906
Alternating series 160f., 1070
Alternative extended Simpson's rule 128
American National Standards Institute (ANSI) 2/x, 2/xiii
Amoeba 403
  see also Simplex, method of Nelder and Mead
Amplification factor 828, 830, 832, 840, 845f.
Amplitude error 831
Analog-to-digital converter 812, 886
Analyticity 195
Analyze/factorize/operate package 64, 824
Anderson-Darling statistic 621
Andrew's sine 697
Annealing, method of simulated 387f., 436ff., 1219ff.
  assessment 447
  for continuous variables 437, 443ff., 1222
  schedule 438
  thermodynamic analogy 437
  traveling salesman problem 438ff., 1219ff.
ANSI (American National Standards Institute) 2/x, 2/xiii
Antonov-Saleev variant of Sobol' sequence 300, 1160
any() intrinsic function 945, 948
APL (computer language) 2/xi
Apple 1/xxiii
  Macintosh 2/xix, 4, 886
Approximate inverse of matrix 49
Approximation of functions 99, 1043
  by Chebyshev polynomials 185f., 513, 1076ff.
  Padé approximant 194ff., 1080f.
  by rational functions 197ff., 1081f.
  by wavelets 594f., 782
  see also Fitting
Argument
  keyword 2/xiv, 947f., 1341
  optional 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
Argument checking 994f., 1086, 1090, 1092, 1370f.

Arithmetic
arbitrary precision 881, 906ff., 1352ff.
floating point 881, 1343
IEEE standard 276, 882, 1343
rounding 882, 1343
Arithmetic coding 881, 902ff., 1349ff.
Arithmetic-geometric mean (AGM) method 906
Arithmetic-if statement 2/xi
Arithmetic progression 971f., 996, 1072, 1127, 1365, 1371f.
Array 953ff.
allocatable 938, 941, 952ff., 1197, 1212, 1266, 1293, 1306, 1336
allocated with pointer 941
allocation 953
array manipulation functions 950
array sections 939, 941, 943ff.
of arrays 2/xii, 956, 1336
associated pointer 953f.
assumed-shape 942
automatic 938, 954, 1197, 1212, 1336
centered subarray of 113
conformable to a scalar 942f., 965, 1094
constructor 2/xii, 968, 971, 1022, 1052, 1055, 1127
copying 991, 1034, 1327f., 1365f.
cumulative product 997f., 1072, 1086, 1375
cumulative sum 997, 1280f., 1365, 1375
deallocation 938, 953f., 1197, 1266, 1293
disassociated pointer 953
extents 938, 949
in Fortran 90 941
increasing storage for 955, 1070, 1302
index loss 967f.
index table 1173ff.
indices 942
inquiry functions 948ff.
intrinsic procedures 2/xiii, 948ff.
of length 0 944
of length 1 949
location of first “true” 993, 1041, 1369
location of maximum value 993, 1015, 1017, 1365, 1369
location of minimum value 993, 1369f.
manipulation functions 950, 1247
masked swapping of elements in two arrays 1368
operations on 942, 949, 964ff., 969, 1026, 1040, 1050, 1200, 1326
outer product 949, 1076
parallel features 941ff., 964ff., 985
passing variable number of arguments to function 1022
of pointers forbidden 956, 1337
rank 938, 949
reallocation 955, 992, 1070f., 1365, 1368f.
reduction functions 948ff.
shape 938, 944, 949
size 938
skew sections 945, 985
stride 944
subscript bounds 942
subscript triplet 944
swapping elements of two arrays 991, 1015, 1365ff.
target 938
three-dimensional, in Fortran 90 1248
transformational functions 948ff.
unary and binary functions 949
undefined status 952ff., 961, 1266, 1293
zero-length 944
Array section 2/xiii, 943ff., 960
matches by shape 944
pointer alias 939, 944f., 1286, 1333
skew 2/xii, 945, 960, 985, 1284
vs. eoshift 1078
array_copy() utility function 988, 991, 1034, 1153, 1278, 1328
arth() utility function 972, 974, 988, 996, 1072, 1086, 1127
replaces do-list 968
Artificial viscosity 831, 837
Ascending transformation, elliptic integrals 256
ASCII character set 6, 888, 896, 902
Assembly language 269
assert() utility function 988, 994, 1086, 1090, 1249
assert_eq() utility function 988, 995, 1022
associated() intrinsic function 952f.
Associated Legendre polynomials 246ff., 764, 1122f., 1319
recurrence relation for 247
relation to Legendre polynomials 246
Association, measures of 604, 622ff., 1275
Assumed-shape array 942
Asymptotic series 161
exponential integral 218
Attenuation factors 583, 1261
Autocorrelation 492
in linear prediction 558
use of FFT 538f., 1254
Wiener-Khinchin theorem 492, 566f.
AUTODIN-II polynomial 890
Automatic array 938, 954, 1197, 1212, 1336
specifying size of 938, 954
Automatic deallocation 2/xv, 961
Autonomous differential equations 729f.
Autoregressive model (AR) *see* Maximum entropy method (MEM)
Average deviation of distribution 605, 1269
Averaging kernel, in Backus-Gilbert method 807

Backsubstitution 33ff., 39, 42, 92, 1017
in band diagonal matrix 46, 1021
in Cholesky decomposition 90, 1039
complex equations 41
direct for computing $\mathbf{A}^{-1} \cdot \mathbf{B}$ 40
with QR decomposition 93, 1040
relaxation solution of boundary value problems 755, 1316
in singular value decomposition 56, 1022f.
Backtracking 419
in quasi-Newton methods 376f., 1195
Backus-Gilbert method 806ff.
Backus, John 2/x
Backward deflation 363

Bader-Deuflhard method 730, 735, 1310f.
Bairstow's method 364, 370, 1193
Balancing 476f., 1230f.
Band diagonal matrix 42ff., 1019
  backsubstitution 46, 1021
  LU decomposition 45, 1020
  multiply by vector 44, 1019
  storage 44, 1019
Band-pass filter 551, 554f.
  wavelets 584, 592f.
Bandwidth limited function 495
Bank accounts, checksum for 894
Bar codes, checksum for 894
Bartlett window 547, 1254ff.
Base case, of recursive procedure 958
Base of representation 19, 882, 1343
BASIC, Numerical Recipes in 1, 2/x, 2/xviii
Basis functions in general linear least squares 665
Bayes' Theorem 810
Bayesian
  approach to inverse problems 799, 810f., 816f.
  contrasted with frequentist 810
  vs. historic maximum entropy method 816f.
  views on straight line fitting 664
Bays' shuffle 270
Bernoulli number 132
Bessel functions 223ff., 234ff., 936, 1101ff.
  asymptotic form 223f., 229f.
  complex 204
  continued fraction 234, 239
  double precision 223
  fractional order 223, 234ff., 1115ff.
  Miller's algorithm 175, 228, 1106
  modified 229ff.
  modified, fractional order 239ff.
  modified, normalization formula 232, 240
  modified, routines for 230ff., 1109ff.
  normalization formula 175
  parallel computation of 1107ff.
  recurrence relation 172, 224, 232, 234
  reflection formulas 236
  reflection formulas, modified functions 241
  routines for 225ff., 236ff., 1101ff.
  routines for modified functions 241ff., 1118
  series for 160, 223
  series for K_ν 241
  series for Y_ν 235
  spherical 234, 245, 1121f.
  turning point 234
  Wronskian 234, 239
Best-fit parameters 650, 656, 660, 698, 1285ff.
  see also Fitting
Beta function 206ff., 1089
  incomplete *see* Incomplete beta function
BFGS algorithm *see* Broyden-Fletcher-Goldfarb-Shanno algorithm
Bias, of exponent 19
Bias, removal in linear prediction 563
Biconjugacy 77
Biconjugate gradient method
  elliptic partial differential equations 824
  preconditioning 78f., 824, 1037
  for sparse system 77, 599, 1034ff.
Bicubic interpolation 118f., 1049f.
Bicubic spline 120f., 1050f.
Big-endian 293
Bilinear interpolation 117
Binary constant, initialization 959
Binomial coefficients 206ff., 1087f.
  recurrences for 209
Binomial probability function 208
  cumulative 222f.
  deviates from 281, 285f., 1155
Binormal distribution 631, 690
Biorthogonality 77
Bisection 111, 359, 1045f.
  compared to minimum bracketing 390ff.
  minimum finding with derivatives 399
  root finding 343, 346f., 352f., 390, 469, 1184f.
BISYNCH 890
Bit 18
  manipulation functions *see* Bitwise logical functions
  reversal in fast Fourier transform (FFT) 499f., 525
bit_size() intrinsic function 951
Bitwise logical functions 2/xiii, 17, 287, 890f., 951
Block-by-block method 788
Block of statements 7
Bode's rule 126
Boltzmann probability distribution 437
Boltzmann's constant 437
Bootstrap method 686f.
Bordering method for Toeplitz matrix 85f.
Borwein and Borwein method for π 906, 1357
Boundary 155f., 425f., 745
Boundary conditions
  for differential equations 701f.
  initial value problems 702
  in multigrid method 868f.
  partial differential equations 508, 819ff., 848ff.
  for spheroidal harmonics 764
  two-point boundary value problems 702, 745ff., 1314ff.
Boundary value problems *see* Differential equations; Elliptic partial differential equations; Two-point boundary value problems
Box-Muller algorithm for normal deviate 279f., 1152
Bracketing
  of function minimum 343, 390ff., 402, 1201f.
  of roots 341, 343ff., 353f., 362, 364, 369, 390, 1183f.
Branch cut, for hypergeometric function 203
Branching 9
Break iteration 14
Brenner, N.M. 500, 517

Brent's method
minimization 389, 395ff., 660f., 1204ff., 1286
minimization, using derivative 389, 399, 1205
root finding 341, 349, 660f., 1188f., 1286
Broadcast (parallel capability) 965ff.
Broyden-Fletcher-Goldfarb-Shanno algorithm 390, 418ff., 1215
Broyden's method 373, 382f., 386, 1199f.
singular Jacobian 386
btest() intrinsic function 951
Bubble sort 321, 1168
Bugs 4
in compilers 1/xvii
how to report 1/iv, 2/iv
Bulirsch-Stoer
algorithm for rational function interpolation 105f., 1043
method (differential equations) 202, 263, 702f., 706, 716, 718ff., 726, 740, 1138, 1303ff.
method (differential equations), stepsize control 719, 726
for second order equations 726, 1307
Burg's LP algorithm 561, 1256
Byte 18

C (programming language) 13, 2/viii
and case construct 1010
Numerical Recipes in 1, 2/x, 2/xvii
C++ 1/xiv, 2/viii, 2/xvi, 7f.
class templates 1083, 1106
Calendar algorithms 1f., 13ff., 1010ff.
Calibration 653
Capital letters in programs 3, 937
Cards, sorting a hand of 321
Carlson's elliptic integrals 255f., 1128ff.
case construct 2/xiv, 1010
trapping errors 1036
Cash-Karp parameters 710, 1299f.
Cauchy probability distribution *see* Lorentzian probability distribution
Cauchy problem for partial differential equations 818f.
Cayley's representation of $\exp(-iHt)$ 844
CCITT (Comité Consultatif International Télégraphique et Téléphonique) 889f., 901
CCITT polynomial 889f.
ceiling() intrinsic function 947
Center of mass 295ff.
Central limit theorem 652f.
Central tendency, measures of 604ff., 1269
Change of variable
in integration 137ff., 788, 1056ff.
in Monte Carlo integration 298
in probability distribution 279
Character functions 952
Character variables, in Fortran 90 1183
Characteristic polynomial
digital filter 554
eigensystems 449, 469
linear prediction 559
matrix with a specified 368, 1193
of recurrence relation 175
Characteristics of partial differential equations 818
Chebyshev acceleration in successive overrelaxation (SOR) 859f., 1332
Chebyshev approximation 84, 124, 183, 184ff., 1076ff.
Clenshaw-Curtis quadrature 190
Clenshaw's recurrence formula 187, 1076
coefficients for 185f., 1076
contrasted with Padé approximation 195
derivative of approximated function 183, 189, 1077f.
economization of series 192f., 195, 1080
for error function 214, 1095
even function 188
and fast cosine transform 513
gamma functions 236, 1118
integral of approximated function 189, 1078
odd function 188
polynomial fits derived from 191, 1078
rational function 197ff., 1081f.
Remes exchange algorithm for filter 553
Chebyshev polynomials 184ff., 1076ff.
continuous orthonormality 184
discrete orthonormality 185
explicit formulas for 184
formula for x^k in terms of 193, 1080
Check digit 894, 1345f.
Checksum 881, 888
cyclic redundancy (CRC) 888ff., 1344f.
Cherry, sundae without a 809
Chi-by-eye 651
Chi-square fitting *see* Fitting; Least squares fitting
Chi-square probability function 209ff., 215, 615, 654, 798, 1272
as boundary of confidence region 688f.
related to incomplete gamma function 215
Chi-square test 614f.
for binned data 614f., 1272
chi-by-eye 651
and confidence limit estimation 688f.
for contingency table 623ff., 1275
degrees of freedom 615f.
for inverse problems 797
least squares fitting 653ff., 1285
nonlinear models 675ff., 1292
rule of thumb 655
for straight line fitting 655ff., 1285
for straight line fitting, errors in both coordinates 660, 1286ff.
for two binned data sets 616, 1272
unequal size samples 617
Chip rate 290
Chirp signal 556
Cholesky decomposition 89f., 423, 455, 1038
backsubstitution 90, 1039
operation count 90
pivoting 90
solution of normal equations 668
Circulant 585
Class, data type 7
Clenshaw-Curtis quadrature 124, 190, 512f.

Clenshaw's recurrence formula 176f., 191, 1078
  for Chebyshev polynomials 187, 1076
  stability 176f.
Clocking errors 891
CM computers (Thinking Machines Inc.) 964
CM Fortran 2/xv
cn function 261, 1137f.
Coarse-grid correction 864f.
Coarse-to-fine operator 864, 1337
Coding
  arithmetic 902ff., 1349ff.
  checksums 888, 1344
  decoding a Huffman-encoded message 900, 1349
  Huffman 896f., 1346ff.
  run-length 901
  variable length code 896, 1346ff.
  Ziv-Lempel 896
  see also Arithmetic coding; Huffman coding
Coefficients
  binomial 208, 1087f.
  for Gaussian quadrature 140ff., 1059ff.
  for Gaussian quadrature, nonclassical weight function 151ff., 788f., 1064
  for quadrature formulas 125ff., 789, 1328
Cohen, Malcolm 2/xiv
Column degeneracy 22
Column operations on matrix 29, 31f.
Column totals 624
Combinatorial minimization *see* Annealing
Comité Consultatif International Télégraphique et Téléphonique (CCITT) 889f., 901
Common block
  obsolescent 2/xif.
  superseded by internal subprogram 957, 1067
  superseded by module 940, 953, 1298, 1320, 1322, 1324, 1330
Communication costs, in parallel processing 969, 981, 1250
Communication theory, use in adaptive integration 721
Communications protocol 888
Comparison function for rejection method 281
Compilers 964, 1364
  CM Fortran 968
  DEC (Digital Equipment Corp.) 2/viii
  IBM (International Business Machines) 2/viii
  Microsoft Fortran PowerStation 2/viii
  NAG (Numerical Algorithms Group) 2/viii, 2/xiv
  for parallel supercomputers 2/viii
Complementary error function 1094f.
  see Error function
Complete elliptic integral *see* Elliptic integrals
Complex arithmetic 171f.
  avoidance of in path integration 203
  cubic equations 179f.
  for linear equations 41
  quadratic equations 178
Complex error function 252
Complex plane
  fractal structure for Newton's rule 360f.
  path integration for function evaluation 201ff., 263, 1138
  poles in 105, 160, 202f., 206, 554, 566, 718f.
Complex systems of linear equations 41f.
Compression of data 596f.
Concordant pair for Kendall's tau 637, 1281
Condition number 53, 78
Confidence level 687, 691ff.
Confidence limits
  bootstrap method 687f.
  and chi-square 688f.
  confidence region, confidence interval 687
  on estimated model parameters 684ff.
  by Monte Carlo simulation 684ff.
  from singular value decomposition (SVD) 693f.
Confluent hypergeometric function 204, 239
Conformable arrays 942f., 1094
Conjugate directions 408f., 414ff., 1210
Conjugate gradient method
  biconjugate 77, 1034
  compared to variable metric method 418
  elliptic partial differential equations 824
  for minimization 390, 413ff., 804, 815, 1210, 1214
  minimum residual method 78
  preconditioner 78f., 1037
  for sparse system 77ff., 599, 1034
  and wavelets 599
Conservative differential equations 726, 1307
Constrained linear inversion method 799ff.
Constrained linear optimization *see* Linear programming
Constrained optimization 387
Constraints, deterministic 804ff.
Constraints, linear 423
CONTAINS statement 954, 957, 1067, 1134, 1202
Contingency coefficient C 625, 1275
Contingency table 622ff., 638, 1275f.
  statistics based on chi-square 623ff., 1275
  statistics based on entropy 626ff., 1275f.
Continued fraction 163ff.
  Bessel functions 234
  convergence criterion 165
  equivalence transformation 166
  evaluation 163ff.
  evaluation along with normalization condition 240
  even and odd parts 166, 211, 216
  even part 249, 251
  exponential integral 216
  Fresnel integral 248f.
  incomplete beta function 219f., 1099f.
  incomplete gamma function 211, 1092f.
  Lentz's method 165, 212
  modified Lentz's method 165
  Pincherle's theorem 175
  ratio of Bessel functions 239
  rational function approximation 164, 211, 219f.
  recurrence for evaluating 164f.

and recurrence relation 175
sine and cosine integrals 250f.
Steed's method 164f.
tangent function 164
typography for 163
Continuous variable (statistics) 623
Control structures 7ff., 2/xiv
bad 15
named 959, 1219, 1305
Convergence
accelerated, for series 160ff., 1070
of algorithm for pi 906
criteria for 347, 392, 404, 483, 488, 679, 759
eigenvalues accelerated by shifting 470f.
golden ratio 349, 399
of golden section search 392f.
of Levenberg-Marquardt method 679
linear 346, 393
of QL method 470f.
quadratic 49, 351, 356, 409f., 419, 906
rate 346f., 353, 356
recurrence relation 175
of Ridders' method 351
series vs. continued fraction 163f.
and spectral radius 856ff., 862
Conversion intrinsic functions 946f.
Convex sets, use in inverse problems 804
Convolution
denoted by asterisk 492
finite impulse response (FIR) 531
of functions 492, 503f.
of large data sets 536f.
for multiple precision arithmetic 909, 1354
multiplication as 909, 1354
necessity for optimal filtering 535
overlap-add method 537
overlap-save method 536f.
and polynomial interpolation 113
relation to wavelet transform 585
theorem 492, 531ff., 546
theorem, discrete 531ff.
treatment of end effects 533
use of FFT 523, 531ff., 1253
wraparound problem 533
Cooley-Tukey FFT algorithm 503, 1250
parallel version 1239f.
Co-processor, floating point 886
Copyright rules 1/xx, 2/xix
Cornwell-Evans algorithm 816
Corporate promotion ladder 328
Corrected two-pass algorithm 607, 1269
Correction, in multigrid method 863
Correlation coefficient (linear) 630ff., 1276
Correlation function 492
autocorrelation 492, 539, 558
and Fourier transforms 492
theorem 492, 538
treatment of end effects 538f.
using FFT 538f., 1254
Wiener-Khinchin theorem 492, 566f.
Correlation, statistical 603f., 622
Kendall's tau 634, 637ff., 1279
linear correlation coefficient 630ff., 658, 1276
linear related to least square fitting 630, 658
nonparametric or rank statistical 633ff., 1277
among parameters in a fit 657, 667, 670
in random number generators 268
Spearman rank-order coefficient 634f., 1277
sum squared difference of ranks 634, 1277
Cosine function, recurrence 172
Cosine integral 248, 250ff., 1125f.
continued fraction 250
routine for 251f., 1125
series 250
Cosine transform *see* Fast Fourier transform (FFT); Fourier transform
Coulomb wave function 204, 234
count() intrinsic function 948
Courant condition 829, 832ff., 836
multidimensional 846
Courant-Friedrichs-Lewy stability criterion *see* Courant condition
Covariance
a priori 700
in general linear least squares 667, 671, 1288ff.
matrix, by Cholesky decomposition 91, 667
matrix, of errors 796, 808
matrix, is inverse of Hessian matrix 679
matrix, when it is meaningful 690ff.
in nonlinear models 679, 681, 1292
relation to chi-square 690ff.
from singular value decomposition (SVD) 693f.
in straight line fitting 657
cpu_time() intrinsic function (Fortran 95) 961
CR method *see* Cyclic reduction (CR)
Cramer's V 625, 1275
Crank-Nicholson method 840, 844, 846
Cray computers 964
CRC (cyclic redundancy check) 888ff., 1344f.
CRC-12 890
CRC-16 polynomial 890
CRC-CCITT 890
Creativity, essay on 9
Critical (Nyquist) sampling 494, 543
Cross (denotes matrix outer product) 66
Crosstabulation analysis 623
see also Contingency table
Crout's algorithm 36ff., 45, 1017
cshift() intrinsic function 950
communication bottleneck 969
Cubic equations 178ff., 360
Cubic spline interpolation 107ff., 1044f.
see also Spline
cumprod() utility function 974, 988, 997, 1072, 1086
cumsum() utility function 974, 989, 997, 1280, 1305
Cumulant, of a polynomial 977, 999, 1071f., 1192

Cumulative binomial distribution 222f.
Cumulative Poisson function 214
related to incomplete gamma function 214
Curvature matrix *see* Hessian matrix
cycle statement 959, 1219
Cycle, in multigrid method 865
Cyclic Jacobi method 459, 1225
Cyclic reduction (CR) 848f., 852ff.
linear recurrences 974
tridiagonal systems 976, 1018
Cyclic redundancy check (CRC) 888ff., 1344f.
Cyclic tridiagonal systems 67, 1030

D.C. (direct current) 492
Danielson-Lanczos lemma 498f., 525, 1235ff.
DAP Fortran 2/xi
Data
assigning keys to 889
continuous vs. binned 614
entropy 626ff., 896, 1275
essay on 603
fitting 650ff., 1285ff.
fraudulent 655
glitches in 653
iid (independent and identically distributed) 686
modeling 650ff., 1285ff.
serial port 892
smoothing 604, 644ff., 1283f.
statistical tests 603ff., 1269ff.
unevenly or irregularly sampled 569, 574, 648f., 1258ff.
use of CRCs in manipulating 889
windowing 545ff., 1254
see also Statistical tests
Data compression 596f., 881
arithmetic coding 902ff., 1349ff.
cosine transform 513
Huffman coding 896f., 902, 1346ff.
linear predictive coding (LPC) 563ff.
lossless 896
Data Encryption Standard (DES) 290ff., 1144, 1147f., 1156ff.
Data hiding 956ff., 1209, 1293, 1296
Data parallelism 941, 964ff., 985
DATA statement 959
for binary, octal, hexadecimal constants 959
repeat count feature 959
superseded by initialization expression 943, 959, 1127
Data type 18, 936
accuracy parameters 1362f.
character 1183
derived 2/xiii, 937, 1030, 1336, 1346
derived, for array of arrays 956, 1336
derived, initialization 2/xv
derived, for Numerical Recipes 1361
derived, storage allocation 955
DP (double precision) 1361f.
DPC (double precision complex) 1361
I1B (1 byte integer) 1361
I2B (2 byte integer) 1361
I4B (4 byte integer) 1361
intrinsic 937
LGT (default logical type) 1361
nrtype.f90 1361f.
passing complex as real 1140
SP (single precision) 1361f.
SPC (single precision complex) 1361
user-defined 1346
DAUB4 584ff., 588, 590f., 594, 1264f.
DAUB6 586
DAUB12 598
DAUB20 590f., 1265
Daubechies wavelet coefficients 584ff., 588, 590f., 594, 598, 1264ff.
Davidon-Fletcher-Powell algorithm 390, 418ff., 1215
Dawson's integral 252ff., 600, 1127f.
approximation for 252f.
routine for 253f., 1127
dble() intrinsic function (deprecated) 947
deallocate statement 938f., 953f., 1197, 1266, 1293
Deallocation, of allocatable array 938, 953f., 1197, 1266, 1293
Debugging 8
DEC (Digital Equipment Corp.) 1/xxiii, 2/xix, 886
Alpha AXP 2/viii
Fortran 90 compiler 2/viii
quadruple precision option 1362
VAX 4
Decomposition *see* Cholesky decomposition; LU decomposition; QR decomposition; Singular value decomposition (SVD)
Deconvolution 535, 540, 1253
see also Convolution; Fast Fourier transform (FFT); Fourier transform
Defect, in multigrid method 863
Deferred approach to the limit *see* Richardson's deferred approach to the limit
Deflation
of matrix 471
of polynomials 362ff., 370f., 977
Degeneracy of linear algebraic equations 22, 53, 57, 670
Degenerate kernel 785
Degenerate minimization principle 795
Degrees of freedom 615f., 654, 691
Dekker, T.J. 353
Demonstration programs 3, 936
Deprecated features
common block 2/xif., 940, 953, 957, 1067, 1298, 1320, 1322, 1324, 1330
dble() intrinsic function 947
EQUIVALENCE statement 2/xif., 1161, 1286
statement function 1057, 1256
Derivatives
computation via Chebyshev approximation 183, 189, 1077f.
computation via Savitzky-Golay filters 183, 645
matrix of first partial *see* Jacobian determinant
matrix of second partial *see* Hessian matrix

numerical computation 180ff., 379, 645, 732, 750, 771, 1075, 1197, 1309
of polynomial 167, 978, 1071f.
use in optimization 388f., 399, 1205ff.
Derived data type *see* Data type, derived
DES *see* Data Encryption Standard
Descending transformation, elliptic integrals 256
Descent direction 376, 382, 419
Descriptive statistics 603ff., 1269ff.
see also Statistical tests
Design matrix 645, 665, 795, 801, 1082
Determinant 25, 41
Deviates, random *see* Random deviates
DFP algorithm *see* Davidon-Fletcher-Powell algorithm
diagadd() utility function 985, 989, 1004
diagmult() utility function 985, 989, 1004, 1294
Diagonal dominance 43, 679, 780, 856
Difference equations, finite *see* Finite difference equations (FDEs)
Difference operator 161
Differential equations 701ff., 1297ff.
accuracy vs. stability 704, 729
Adams-Bashforth-Moulton schemes 741
adaptive stepsize control 703, 708ff., 719, 726, 731, 737, 742f., 1298ff., 1303ff., 1308f., 1311ff.
algebraically difficult sets 763
backward Euler's method 729
Bader-Deuflhard method for stiff 730, 735, 1310f.
boundary conditions 701f., 745ff., 749, 751f., 771, 1314ff.
Bulirsch-Stoer method 202, 263, 702, 706, 716, 718ff., 740, 1138, 1303
Bulirsch-Stoer method for conservative equations 726, 1307
comparison of methods 702f., 739f., 743
conservative 726, 1307
danger of too small stepsize 714
eigenvalue problem 748, 764ff., 770ff., 1319ff.
embedded Runge-Kutta method 709f., 731, 1298, 1308
equivalence of multistep and multivalue methods 743
Euler's method 702, 704, 728f.
forward Euler's method 728
free boundary problem 748, 776
high-order implicit methods 730ff., 1308ff.
implicit differencing 729, 740, 1308
initial value problems 702
internal boundary conditions 775ff.
internal singular points 775ff.
interpolation on right-hand sides 111
Kaps-Rentrop method for stiff 730, 1308
local extrapolation 709
modified midpoint method 716f., 719, 1302f.
multistep methods 740ff.
multivalue methods 740
order of method 704f., 719
path integration for function evaluation 201ff., 263, 1138
predictor-corrector methods 702, 730, 740ff.
reduction to first-order sets 701, 745
relaxation method 746f., 753ff., 1316ff.
relaxation method, example of 764ff., 1319ff.
r.h.s. independent of x 729f.
Rosenbrock methods for stiff 730, 1308f.
Runge-Kutta method 702, 704ff., 708ff., 731, 740, 1297f., 1308
Runge-Kutta method, high-order 705, 1297
Runge-Kutta-Fehlberg method 709ff., 1298
scaling stepsize to required accuracy 709
second order 726, 1307
semi-implicit differencing 730
semi-implicit Euler method 730, 735f.
semi-implicit extrapolation method 730, 735f., 1311ff.
semi-implicit midpoint rule 735f., 1310f.
shooting method 746, 749ff., 1314ff.
shooting method, example 770ff., 1321ff.
similarity to Volterra integral equations 786
singular points 718f., 751, 775ff., 1315f., 1323ff.
step doubling 708f.
stepsize control 703, 708ff., 719, 726, 731, 737, 742f., 1298, 1303ff., 1308f.
stiff 703, 727ff., 1308ff.
stiff methods compared 739
Stoermer's rule 726, 1307
see also Partial differential equations; Two-point boundary value problems
Diffusion equation 818, 838ff., 855
Crank-Nicholson method 840, 844, 846
Forward Time Centered Space (FTCS) 839ff., 855
implicit differencing 840
multidimensional 846
Digamma function 216
Digital filtering *see* Filter
Dihedral group D_5 894
dim optional argument 948
Dimensional expansion 965ff.
Dimensions (units) 678
Diminishing increment sort 322, 1168
Dirac delta function 284, 780
Direct method *see* Periodogram
Direct methods for linear algebraic equations 26, 1014
Direct product *see* Outer product of matrices
Direction of largest decrease 410f.
Direction numbers, Sobol's sequence 300
Direction-set methods for minimization 389, 406f., 1210ff.
Dirichlet boundary conditions 820, 840, 850, 856, 858
Disclaimer of warranty 1/xx, 2/xvii
Discordant pair for Kendall's tau 637, 1281
Discrete convolution theorem 531ff.

Discrete Fourier transform (DFT) 495ff., 1235ff.
as approximate continuous transform 497
see also Fast Fourier transform (FFT)
Discrete optimization 436ff., 1219ff.
Discriminant 178, 457
Diskettes
are ANSI standard 3
how to order 1/xxi, 2/xvii
Dispersion 831
DISPO *see* Savitzky-Golay filters
Dissipation, numerical 830
Divergent series 161
Divide and conquer algorithm 1226, 1229
Division
complex 171
multiple precision 910f., 1356
of polynomials 169, 362, 370, 1072
dn function 261, 1137f.
Do-list, implied 968, 971, 1127
Do-loop 2/xiv
Do-until iteration 14
Do-while iteration 13
Dogleg step methods 386
Domain of integration 155f.
Dominant solution of recurrence relation 174
Dot (denotes matrix multiplication) 23
dot_product() intrinsic function 945, 949, 969, 1216
Double exponential error distribution 696
Double precision
converting to 1362
as refuge of scoundrels 882
use in iterative improvement 47, 1022
Double root 341
Downhill simplex method *see* Simplex, method of Nelder and Mead
DP, defined 937
Driver programs 3
Dual viewpoint, in multigrid method 875
Duplication theorem, elliptic integrals 256
DWT (discrete wavelet transform) *see* Wavelet transform
Dynamical allocation of storage 2/xiii, 869, 938, 941f., 953ff., 1327, 1336
garbage collection 956
increasing 955, 1070, 1302

Eardley, D.M. 338
EBCDIC 890
Economization of power series 192f., 195, 1080
Eigensystems 449ff., 1225ff.
balancing matrix 476f., 1230f.
bounds on eigenvalues 50
calculation of few eigenvalues 454, 488
canned routines 454f.
characteristic polynomial 449, 469
completeness 450
defective 450, 476, 489
deflation 471
degenerate eigenvalues 449ff.
elimination method 453, 478, 1231
factorization method 453
fast Givens reduction 463
generalized eigenproblem 455
Givens reduction 462f.
Hermitian matrix 475
Hessenberg matrix 453, 470, 476ff., 488, 1232
Householder transformation 453, 462ff., 469, 473, 475, 478, 1227f., 1231
ill-conditioned eigenvalues 477
implicit shifts 472ff., 1228f.
and integral equations 779, 785
invariance under similarity transform 452
inverse iteration 455, 469, 476, 487ff., 1230
Jacobi transformation 453, 456ff., 462, 475, 489, 1225f.
left eigenvalues 451
list of tasks 454f.
multiple eigenvalues 489
nonlinear 455
nonsymmetric matrix 476ff., 1230ff.
operation count of balancing 476
operation count of Givens reduction 463
operation count of Householder reduction 467
operation count of inverse iteration 488
operation count of Jacobi method 460
operation count of QL method 470, 473
operation count of QR method for Hessenberg matrices 484
operation count of reduction to Hessenberg form 479
orthogonality 450
parallel algorithms 1226, 1229
polynomial roots and 368, 1193
QL method 469ff., 475, 488f.
QL method with implicit shifts 472ff., 1228f.
QR method 52, 453, 456, 469ff., 1228
QR method for Hessenberg matrices 480ff., 1232ff.
real, symmetric matrix 150, 467, 785, 1225, 1228
reduction to Hessenberg form 478f., 1231
right eigenvalues 451
shifting eigenvalues 449, 470f., 480
special matrices 454
termination criterion 484, 488
tridiagonal matrix 453, 469ff., 488, 1228
Eigenvalue and eigenvector, defined 449
Eigenvalue problem for differential equations 748, 764ff., 770ff., 1319ff.
Eigenvalues and polynomial root finding 368, 1193
EISPACK 454, 475
Electromagnetic potential 519
ELEMENTAL attribute (Fortran 95) 961, 1084
Elemental functions 2/xiii, 2/xv, 940, 942, 946f., 961, 986, 1015, 1083, 1097f.
Elimination *see* Gaussian elimination
Ellipse in confidence limit estimation 688
Elliptic integrals 254ff., 906
addition theorem 255

Carlson's forms and algorithms 255f., 1128ff.
Cauchy principal value 256f.
duplication theorem 256
Legendre 254ff., 260f., 1135ff.
routines for 257ff., 1128ff.
symmetric form 255
Weierstrass 255
Elliptic partial differential equations 818, 1332ff.
alternating-direction implicit method (ADI) 861f., 906
analyze/factorize/operate package 824
biconjugate gradient method 824
boundary conditions 820
comparison of rapid methods 854
conjugate gradient method 824
cyclic reduction 848f., 852ff.
Fourier analysis and cyclic reduction (FACR) 848ff., 854
Gauss-Seidel method 855, 864ff., 876, 1338, 1341
incomplete Cholesky conjugate gradient method (ICCG) 824
Jacobi's method 855f., 864
matrix methods 824
multigrid method 824, 862ff., 1009, 1334ff.
rapid (Fourier) method 824, 848ff.
relaxation method 823, 854ff., 1332
strongly implicit procedure 824
successive over-relaxation (SOR) 857ff., 862, 866, 1332
elsewhere construct 943
Emacs, GNU 1/xvi
Embedded Runge-Kutta method 709f., 731, 1298, 1308
Encapsulation, in programs 7
Encryption 290, 1156
enddo statement 12, 17
Entropy 896
of data 626ff., 811, 1275
EOM (end of message) 902
eoshift() intrinsic function 950
communication bottleneck 969
vector shift argument 1019f.
vs. array section 1078
epsilon() intrinsic function 951, 1189
Equality constraints 423
Equations
cubic 178ff., 360
normal (fitting) 645, 666ff., 800, 1288
quadratic 20, 178
see also Differential equations; Partial differential equations; Root finding
Equivalence classes 337f., 1180
EQUIVALENCE statement 2/xif., 1161, 1286
Equivalence transformation 166
Error
checksums for preventing 891
clocking 891
double exponential distribution 696
local truncation 875
Lorentzian distribution 696f.
in multigrid method 863
nonnormal 653, 690, 694ff.
relative truncation 875
roundoff 180f., 881, 1362
series, advantage of an even 132f., 717, 1362
systematic vs. statistical 653, 1362
truncation 20f., 180, 399, 709, 881, 1362
varieties found by check digits 895
varieties of, in PDEs 831ff.
see also Roundoff error
Error function 213f., 601, 1094f.
approximation via sampling theorem 601
Chebyshev approximation 214, 1095
complex 252
for Fisher's z-transformation 632, 1276
relation to Dawson's integral 252, 1127
relation to Fresnel integrals 248
relation to incomplete gamma function 213
routine for 214, 1094
for significance of correlation 631, 1276
for sum squared difference of ranks 635, 1277
Error handling in programs 2/xii, 2/xvi, 3, 994f., 1036, 1370f.
Estimation of parameters *see* Fitting; Maximum likelihood estimate
Estimation of power spectrum 542ff., 565ff., 1254ff., 1258
Euler equation (fluid flow) 831
Euler-Maclaurin summation formula 132, 135
Euler's constant 216ff., 250
Euler's method for differential equations 702, 704, 728f.
Euler's transformation 160f., 1070
generalized form 162f.
Evaluation of functions *see* Function
Even and odd parts, of continued fraction 166, 211, 216
Even parity 888
Exception handling in programs *see* Error handling in programs
exit statement 959, 1219
Explicit differencing 827
Exponent in floating point format 19, 882, 1343
exponent intrinsic function 1107
Exponential deviate 278, 1151f.
Exponential integral 215ff., 1096f.
asymptotic expansion 218
continued fraction 216
recurrence relation 172
related to incomplete gamma function 215
relation to cosine integral 250
routine for $\mathrm{Ei}(x)$ 218, 1097
routine for $E_n(x)$ 217, 1096
series 216
Exponential probability distribution 570
Extended midpoint rule 124f., 129f., 135, 1054f.
Extended Simpson's rule 128, 788, 790
Extended Simpson's three-eighths rule 789
Extended trapezoidal rule 125, 127, 130ff., 135, 786, 1052ff., 1326
roundoff error 132
Extirpolation (so-called) 574, 1261

Extrapolation 99ff.
in Bulirsch-Stoer method 718ff., 726, 1305ff.
differential equations 702
by linear prediction 557ff., 1256f.
local 709
maximum entropy method as type of 567
polynomial 724, 726, 740, 1305f.
rational function 718ff., 726, 1306f.
relation to interpolation 101
for Romberg integration 134
see also Interpolation
Extremization *see* Minimization

F-distribution probability function 222
F-test for differences of variances 611, 613, 1271
FACR *see* Fourier analysis and cyclic reduction (FACR)
Facsimile standard 901
Factorial
double (denoted "!!") 247
evaluation of 159, 1072, 1086
relation to gamma function 206
routine for 207f., 1086ff.
False position 347ff., 1185f.
Family tree 338
FAS (full approximation storage algorithm) 874, 1339ff.
Fast Fourier transform (FFT) 498ff., 881, 981, 1235f.
alternative algorithms 503f.
as approximation to continuous transform 497
Bartlett window 547, 1254
bit reversal 499f., 525
and Clenshaw-Curtis quadrature 190
column-parallel algorithm 981, 1237ff.
communication bottleneck 969, 981, 1250
convolution 503f., 523, 531ff., 909, 1253, 1354
convolution of large data sets 536f.
Cooley-Tukey algorithm 503, 1250
Cooley-Tukey algorithm, parallel 1239f.
correlation 538f., 1254
cosine transform 190, 511ff., 851, 1245f.
cosine transform, second form 513, 852, 1246
Danielson-Lanczos lemma 498f., 525
data sets not a power of 2 503
data smoothing 645
data windowing 545ff., 1254
decimation-in-frequency algorithm 503
decimation-in-time algorithm 503
discrete autocorrelation 539, 1254
discrete convolution theorem 531ff.
discrete correlation theorem 538
at double frequency 575
effect of caching 982
endpoint corrections 578f., 1261ff.
external storage 525
figures of merit for data windows 548
filtering 551ff.
FIR filter 553
four-step framework 983, 1239
Fourier integrals 577ff., 1261
Fourier integrals, infinite range 583
Hamming window 547
Hann window 547
history 498
IIR filter 553ff.
image processing 803, 805
integrals using 124
inverse of cosine transform 512ff.
inverse of sine transform 511
large data sets 525
leakage 544
memory-local algorithm 528
multidimensional 515ff., 1236f., 1241, 1246, 1251
for multiple precision arithmetic 906
for multiple precision multiplication 909, 1354
number-theoretic transforms 503f.
operation count 498
optimal (Wiener) filtering 539ff., 558
order of storage in 501
parallel algorithms 981ff., 1235ff.
partial differential equations 824, 848ff.
Parzen window 547
periodicity of 497
periodogram 543ff., 566
power spectrum estimation 542ff., 1254ff.
for quadrature 124
of real data in 2D and 3D 519ff., 1248f.
of real functions 504ff., 519ff., 1242f., 1248f.
related algorithms 503f.
row-parallel algorithm 981, 1235f.
Sande-Tukey algorithm 503
sine transform 508ff., 850, 1245
Singleton's algorithm 525
six-step framework 983, 1240
square window 546, 1254
timing 982
treatment of end effects in convolution 533
treatment of end effects in correlation 538f.
Tukey's trick for frequency doubling 575
use in smoothing data 645
used for Lomb periodogram 574, 1259
variance of power spectrum estimate 544f., 549
virtual memory machine 528
Welch window 547, 1254
Winograd algorithms 503
see also Discrete Fourier transform (DFT); Fourier transform; Spectral density
Faure sequence 300
Fax (facsimile) Group 3 standard 901
Feasible vector 424
FFT *see* Fast Fourier transform (FFT)
Field, in data record 329
Figure-of-merit function 650
Filon's method 583
Filter 551ff.
acausal 552
bilinear transformation method 554
causal 552, 644

characteristic polynomial 554
data smoothing 644f., 1283f.
digital 551ff.
DISPO 644
by fast Fourier transform (FFT) 523, 551ff.
finite impulse response (FIR) 531, 552
homogeneous modes of 554
infinite impulse response (IIR) 552ff., 566
Kalman 700
linear 552ff.
low-pass for smoothing 644ff., 1283f.
nonrecursive 552
optimal (Wiener) 535, 539ff., 558, 644
quadrature mirror 585, 593
realizable 552, 554f.
recursive 552ff., 566
Remes exchange algorithm 553
Savitzky-Golay 183, 644ff., 1283f.
stability of 554f.
in the time domain 551ff.
Fine-to-coarse operator 864, 1337
Finite difference equations (FDEs) 753, 763, 774
alternating-direction implicit method (ADI) 847, 861f.
art not science 829
Cayley's form for unitary operator 844
Courant condition 829, 832ff., 836
Courant condition (multidimensional) 846
Crank-Nicholson method 840, 844, 846
eigenmodes of 827f.
explicit vs. implicit schemes 827
forward Euler 826f.
Forward Time Centered Space (FTCS) 827ff., 839ff., 843, 855
implicit scheme 840
Lax method 828ff., 836
Lax method (multidimensional) 845f.
mesh drifting instability 834f.
numerical derivatives 181
partial differential equations 821ff.
in relaxation methods 753ff.
staggered leapfrog method 833f.
two-step Lax-Wendroff method 835ff.
upwind differencing 832f., 837
see also Partial differential equations
Finite element methods, partial differential equations 824
Finite impulse response (FIR) 531
Finkelstein, S. 1/xvi, 2/ix
FIR (finite impulse response) filter 552
Fisher's z-transformation 631f., 1276
Fitting 650ff., 1285ff.
basis functions 665
by Chebyshev approximation 185f., 1076
chi-square 653ff., 1285ff.
confidence levels related to chi-square values 691ff.
confidence levels from singular value decomposition (SVD) 693f.
confidence limits on fitted parameters 684ff.
covariance matrix not always meaningful 651, 690
degeneracy of parameters 674
an exponential 674
freezing parameters in 668, 700
Gaussians, a sum of 682, 1294
general linear least squares 665ff., 1288, 1290f.
Kalman filter 700
K–S test, caution regarding 621f.
least squares 651ff., 1285
Legendre polynomials 674, 1291f.
Levenberg-Marquardt method 678ff., 816, 1292f.
linear regression 655ff., 1285ff.
maximum likelihood estimation 652f., 694ff.
Monte Carlo simulation 622, 654, 684ff.
multidimensional 675
nonlinear models 675ff., 1292f.
nonlinear models, advanced methods 683
nonlinear problems that are linear 674
nonnormal errors 656, 690, 694ff.
polynomial 83, 114, 191, 645, 665, 674, 1078, 1291
by rational Chebyshev approximation 197ff., 1081f.
robust methods 694ff., 1294
of sharp spectral features 566
standard (probable) errors on fitted parameters 657f., 661, 667, 671, 684ff., 1285f., 1288, 1290
straight line 655ff., 667f., 698, 1285ff., 1294ff.
straight line, errors in both coordinates 660ff., 1286ff.
see also Error; Least squares fitting; Maximum likelihood estimate; Robust estimation
Five-point difference star 867
Fixed point format 18
Fletcher-Powell algorithm *see* Davidon-Fletcher-Powell algorithm
Fletcher-Reeves algorithm 390, 414ff., 1214
Floating point co-processor 886
Floating point format 18ff., 882, 1343
care in numerical derivatives 181
IEEE 276, 882, 1343
floor() intrinsic function 948
Flux-conservative initial value problems 825ff.
FMG (full multigrid method) 863, 868, 1334ff.
FOR iteration 9f., 12
forall statement 2/xii, 2/xv, 960, 964, 986
access to associated index 968
skew array sections 985, 1007
Formats of numbers 18ff., 882, 1343
Fortran 9
arithmetic-if statement 2/xi
COMMON block 2/xif., 953, 957
deprecated features 2/xif., 947, 1057, 1161, 1256, 1286
dynamical allocation of storage 869, 1336
EQUIVALENCE statement 2/xif., 1161, 1286
evolution of 2/xivff.
exception handling 2/xii, 2/xvi
filenames 935
Fortran 2000 (planned) 2/xvi

Fortran 95 2/xv, 945, 947, 1084, 1100, 1364
HPF (High-Performance Fortran) 2/xvf.
Numerical Recipes in 2/x, 2/xvii, 1
obsolescent features 2/xif.
side effects 960
see also Fortran 90
Fortran D 2/xv
Fortran 77 1/xix
bit manipulation functions 17
hexadecimal constants 17
Fortran 8x 2/xi, 2/xiii
Fortran 90 3
abstract data types 2/xiii, 1030
all() intrinsic function 945, 948
allocatable array 938, 941, 953ff., 1197, 1212, 1266, 1293, 1306, 1336
allocate statement 938f., 941, 953f., 1197, 1266, 1293, 1306, 1336
allocated() intrinsic function 938, 952ff., 1197, 1266, 1293
any() intrinsic function 945, 948
array allocation and deallocation 953
array of arrays 2/xii, 956, 1336
array constructor 2/xii, 968, 971, 1022, 1052, 1055, 1127
array constructor with implied do-list 968, 971
array extents 938, 949
array features 941ff., 953ff.
array intrinsic procedures 2/xiii, 948ff.
array of length 0 944
array of length 1 949
array manipulation functions 950
array parallel operations 964f.
array rank 938, 949
array reallocation 955
array section 2/xiif., 2/xiii, 939, 941ff., 960, 1078, 1284, 1286, 1333
array shape 938, 949
array size 938, 942
array transpose 981f.
array unary and binary functions 949
associated() intrinsic function 952f.
associated pointer 953f.
assumed-shape array 942
automatic array 938, 954, 1197, 1212, 1336
backwards-compatibility 935, 946
bit manipulation functions 2/xiii, 951
bit_size() intrinsic function 951
broadcasts 965f.
btest() intrinsic function 951
case construct 1010, 1036
case insensitive 937
ceiling() intrinsic function 947
character functions 952
character variables 1183
cmplx function 1125
communication bottlenecks 969, 981, 1250
compatibility with Fortran 77 935, 946
compilers 2/viii, 2/xiv, 1364
compiling 936
conformable arrays 942f., 1094
CONTAINS statement 954, 957, 985, 1067, 1134, 1202
control structure 2/xiv, 959, 1219, 1305
conversion elemental functions 946
count() intrinsic function 948
cshift() intrinsic function 950, 969
cycle statement 959, 1219
data hiding 956ff., 1209
data parallelism 964
DATA statement 959
data types 937, 1336, 1346, 1361
deallocate statement 938f., 953f., 1197, 1266, 1293
deallocating array 938, 953f., 1197, 1266, 1293
defined types 956
deprecated features 947, 1057, 1161, 1256, 1286
derived types 937, 955
dimensional expansion 965ff.
do-loop 2/xiv
dot_product() intrinsic function 945, 949, 969, 1216
dynamical allocation of storage 2/xiii, 938, 941f., 953ff., 1327, 1336
elemental functions 940, 942, 946f., 951, 1015, 1083, 1364
elsewhere construct 943
eoshift() intrinsic function 950, 969, 1019f., 1078
epsilon() intrinsic function 951, 1189
evolution 2/xivff., 959, 987f.
example 936
exit statement 959, 1219
exponent() intrinsic function 1107
floor() intrinsic function 948
Fortran tip icon 1009
garbage collection 956
gather-scatter operations 2/xiif., 969, 981, 984, 1002, 1032, 1034, 1250
generic interface 2/xiii, 1083
generic procedures 939, 1015, 1083, 1094, 1096, 1364
global variables 955, 957, 1210
history 2/xff.
huge() intrinsic function 951
iand() intrinsic function 951
ibclr() intrinsic function 951
ibits() intrinsic function 951
ibset() intrinsic function 951
ieor() intrinsic function 951
IMPLICIT NONE statement 2/xiv, 936
implied do-list 968, 971, 1127
index loss 967f.
initialization expression 943, 959, 1012, 1127
inquiry functions 948
integer model 1144, 1149, 1156
INTENT attribute 1072, 1092
interface 939, 942, 1067, 1084, 1384
internal subprogram 2/xii, 2/xiv, 957, 1057, 1067, 1202f., 1256, 1302
interprocessor communication 969, 981, 1250
intrinsic data types 937

intrinsic procedures 939, 945ff., 987, 1016
ior() intrinsic function 951
ishft() intrinsic function 951
ishftc() intrinsic function 951
ISO (International Standards Organization) 2/xf., 2/xiiif.
keyword argument 2/xiv, 947f., 1341
kind() intrinsic function 951
KIND parameter 937, 946, 1125, 1144, 1192, 1254, 1261, 1284, 1361
language features 935ff.
lbound() intrinsic function 949
lexical comparison 952
linear algebra 969f., 1000ff., 1018f., 1026, 1040, 1200, 1326
linear recurrence 971, 988
linking 936
literal constant 937, 1361
logo for tips 2/viii, 1009
mask 948, 967f., 1006f., 1038, 1102, 1200, 1226, 1305, 1333f., 1368, 1378, 1382
matmul() intrinsic function 945, 949, 969, 1026, 1040, 1050, 1076, 1200, 1216, 1290, 1326
maxexponent() intrinsic function 1107
maxloc() intrinsic function 949, 961, 992f., 1015
maxval() intrinsic function 945, 948, 961, 1016, 1273
memory leaks 953, 956, 1327
memory management 938, 953ff.
merge() intrinsic function 945, 950, 1010, 1094f., 1099f.
Metcalf and Reid (M&R) 935
minloc() intrinsic function 949, 961, 992f.
minval() intrinsic function 948, 961
missing language features 983ff., 987ff.
modularization 956f.
MODULE facility 2/xiii, 936f., 939f., 953f., 957, 1067, 1298, 1320, 1322, 1324, 1330, 1346
MODULE subprograms 940
modulo() intrinsic function 946, 1156
named constant 940, 1012, 1361
named control structure 959, 1219, 1305
nearest() intrinsic function 952, 1146
nested where construct forbidden 943
not() intrinsic function 951
nullify statement 953f., 1070, 1302
numerical representation functions 951
ONLY option 941, 957, 1067
operator overloading 2/xiif.
operator, user-defined 2/xii
optional argument 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
outer product 969f.
overloading 940, 1083, 1102
pack() intrinsic function 945, 950, 964, 969, 991, 1170, 1176, 1178
pack, for selective evaluation 1087
parallel extensions 2/xv, 959ff., 964, 981, 984, 987, 1002, 1032
parallel programming 963ff.
PARAMETER attribute 1012
pointer 2/xiiif., 938f., 941, 944f., 952ff., 1067, 1070, 1197, 1210, 1212, 1266, 1302, 1327, 1336
pointer to function (missing) 1067
portability 963
present() intrinsic function 952
PRIVATE attribute 957, 1067
product() intrinsic function 948
programming conventions 937
PUBLIC attribute 957, 1067
quick start 936
radix() intrinsic function 1231
random_number() intrinsic function 1141, 1143
random_seed() intrinsic function 1141
real() intrinsic function 947, 1125
RECURSIVE keyword 958, 1065, 1067
recursive procedure 2/xiv, 958, 1065, 1067, 1166
reduction functions 948
reshape() intrinsic function 950, 969, 1247
RESULT keyword 958, 1073
SAVE attribute 953f., 958f., 1052, 1070, 1266, 1293
scale() intrinsic function 1107
scatter-with-combine (missing function) 984
scope 956ff.
scoping units 939
select case statement 2/xiv, 1010, 1036
shape() intrinsic function 938, 949
size() intrinsic function 938, 942, 945, 948
skew sections 985
sparse matrix representation 1030
specification statement 2/xiv
spread() intrinsic function 945, 950, 966ff., 969, 1000, 1094, 1290f.
statement functions deprecated 1057
stride (of an array) 944
structure constructor 2/xii
subscript triplet 944
sum() intrinsic function 945, 948, 966
tiny() intrinsic function 952
transformational functions 948
transpose() intrinsic function 950, 960, 969, 981, 1247
tricks 1009, 1072, 1146, 1274, 1278, 1280
truncation elemental functions 946
type checking 1140
ubound() intrinsic function 949
undefined pointer 953
unpack() intrinsic function 950, 964, 969
USE statement 936, 939f., 954, 957, 1067, 1384
utility functions 987ff.
vector subscripts 2/xiif., 969, 981, 984, 1002, 1032, 1034, 1250
visibility 956ff., 1209, 1293, 1296
WG5 technical committee 2/xi, 2/xiii, 2/xvf.
where construct 943, 985, 1060, 1291
X3J3 Committee 2/viii, 2/xff., 2/xv, 947, 959, 964, 968, 990
zero-length array 944

see also Intrinsic procedures
see also Fortran
Fortran 95 947, 959ff.
allocatable variables 961
blocks 960
cpu_time() intrinsic function 961
elemental functions 2/xiii, 2/xv, 940, 961, 986, 1015, 1083f., 1097f.
forall statement 2/xii, 2/xv, 960, 964, 968, 986, 1007
initialization of derived data type 2/xv
initialization of pointer 2/xv, 961
minor changes from Fortran 90 961
modified intrinsic functions 961
nested where construct 2/xv, 960, 1100
pointer association status 961
pointers 961
PURE attribute 2/xv, 960f., 964, 986
SAVE attribute 961
side effects 960
and skew array section 945, 985
see also Fortran
Fortran 2000 2/xvi
Forward deflation 363
Forward difference operator 161
Forward Euler differencing 826f.
Forward Time Centered Space *see* FTCS
Four-step framework, for FFT 983, 1239
Fourier analysis and cyclic reduction (FACR) 848f., 854
Fourier integrals
attenuation factors 583, 1261
endpoint corrections 578f., 1261
tail integration by parts 583
use of fast Fourier transform (FFT) 577ff., 1261ff.
Fourier transform 99, 490ff., 1235ff.
aliasing 495, 569
approximation of Dawson's integral 253
autocorrelation 492
basis functions compared 508f.
contrasted with wavelet transform 584, 594
convolution 492, 503f., 531ff., 909, 1253, 1354
correlation 492, 538f., 1254
cosine transform 190, 511ff., 851, 1245f.
cosine transform, second form 513, 852, 1246
critical sampling 494, 543, 545
definition 490
discrete Fourier transform (DFT) 184, 495ff.
Gaussian function 600
image processing 803, 805
infinite range 583
inverse of discrete Fourier transform 497
method for partial differential equations 848ff.
missing data 569
missing data, fast algorithm 574f., 1259
Nyquist frequency 494ff., 520, 543, 545, 569, 571
optimal (Wiener) filtering 539ff., 558
Parseval's theorem 492, 498, 544
power spectral density (PSD) 492f.
power spectrum estimation by FFT 542ff., 1254ff.
power spectrum estimation by maximum entropy method 565ff., 1258
properties of 491f.
sampling theorem 495, 543, 545, 600
scalings of 491
significance of a peak in 570
sine transform 508ff., 850, 1245
symmetries of 491
uneven sampling, fast algorithm 574f., 1259
unevenly sampled data 569ff., 574, 1258
and wavelets 592f.
Wiener-Khinchin theorem 492, 558, 566f.
see also Fast Fourier transform (FFT); Spectral density
Fractal region 360f.
Fractional step methods 847f.
Fredholm alternative 780
Fredholm equations 779f.
eigenvalue problems 780, 785
error estimate in solution 784
first kind 779
Fredholm alternative 780
homogeneous, second kind 785, 1325
homogeneous vs. inhomogeneous 779f.
ill-conditioned 780
infinite range 789
inverse problems 780, 795ff.
kernel 779f.
nonlinear 781
Nystrom method 782ff., 789, 1325
product Nystrom method 789, 1328ff.
second kind 779f., 782ff., 1325, 1331
with singularities 788, 1328ff.
with singularities, worked example 792, 1328ff.
subtraction of singularity 789
symmetric kernel 785
see also Inverse problems
Frequency domain 490
Frequency spectrum *see* Fast Fourier transform (FFT)
Frequentist, contrasted with Bayesian 810
Fresnel integrals 248ff.
asymptotic form 249
continued fraction 248f.
routine for 249f., 1123
series 248
Friday the Thirteenth 14f., 1011f.
FTCS (forward time centered space) 827ff., 839ff., 843
stability of 827ff., 839ff., 855
Full approximation storage (FAS) algorithm 874, 1339ff.
Full moon 14f., 936, 1011f.
Full multigrid method (FMG) 863, 868, 1334ff.
Full Newton methods, nonlinear least squares 683
Full pivoting 29, 1014
Full weighting 867
Function
Airy 204, 243f., 1121

approximation 99ff., 184ff., 1043, 1076ff.
associated Legendre polynomial 246ff., 764, 1122f., 1319
autocorrelation of 492
bandwidth limited 495
Bessel 172, 204, 223ff., 234, 1101ff., 1115ff.
beta 209, 1089
binomial coefficients 208f., 1087f.
branch cuts of 202f.
chi-square probability 215, 798
complex 202
confluent hypergeometric 204, 239
convolution of 492
correlation of 492
cosine integral 250f., 1123f.
Coulomb wave 204, 234
cumulative binomial probability 222f.
cumulative Poisson 209ff.
Dawson's integral 252ff., 600, 1127f.
digamma 216
elliptic integrals 254ff., 906, 1128ff.
error 213f., 248, 252, 601, 631, 635, 1094f., 1127, 1276f.
evaluation 159ff., 1070ff.
evaluation by path integration 201ff., 263, 1138
exponential integral 172, 215ff., 250, 1096f.
F-distribution probability 222
Fresnel integral 248ff., 1123
gamma 206, 1085
hypergeometric 202f., 263ff., 1138ff.
incomplete beta 219ff., 610, 1098ff., 1269
incomplete gamma 209ff., 615, 654, 657f., 1089ff., 1272, 1285
inverse hyperbolic 178, 255
inverse trigonometric 255
Jacobian elliptic 261, 1137f.
Kolmogorov-Smirnov probability 618f., 640, 1274, 1281
Legendre polynomial 172, 246, 674, 1122, 1291
logarithm 255
modified Bessel 229ff., 1109ff.
modified Bessel, fractional order 239ff., 1118ff.
overloading 1083
parallel evaluation 986, 1009, 1084, 1087, 1090, 1102, 1128, 1134
path integration to evaluate 201ff.
pathological 99f., 343
Poisson cumulant 214
representations of 490
routine for plotting a 342, 1182
sine and cosine integrals 248, 250ff., 1125f.
sn, dn, cn 261, 1137f.
spherical harmonics 246ff., 1122
spheroidal harmonic 764ff., 770ff., 1319ff., 1323ff.
Student's probability 221f.
variable number of arguments 1022
Weber 204
Functional iteration, for implicit equations 740f.
FWHM (full width at half maximum) 548f.

Gamma deviate 282f., 1153f.
Gamma function 206ff., 1085
  incomplete *see* Incomplete gamma function
Garbage collection 956
Gather-scatter operations 2/xiif., 984, 1002, 1032, 1034
  communication bottleneck 969, 981, 1250
  many-to-one 984, 1002, 1032, 1034
Gauss-Chebyshev integration 141, 144, 512f.
Gauss-Hermite integration 144, 789
  abscissas and weights 147, 1062
  normalization 147
Gauss-Jacobi integration 144
  abscissas and weights 148, 1063
Gauss-Jordan elimination 27ff., 33, 64, 1014f.
  operation count 34, 39
  solution of normal equations 667, 1288
  storage requirements 30
Gauss-Kronrod quadrature 154
Gauss-Laguerre integration 144, 789, 1060
Gauss-Legendre integration 145f., 1059
  see also Gaussian integration
Gauss-Lobatto quadrature 154, 190, 512
Gauss-Radau quadrature 154
Gauss-Seidel method (relaxation) 855, 857, 864ff., 1338
  nonlinear 876, 1341
Gauss transformation 256
Gaussian (normal) distribution 267, 652, 798
  central limit theorem 652f.
  deviates from 279f., 571, 1152
  kurtosis of 606
  multivariate 690
  semi-invariants of 608
  tails compared to Poisson 653
  two-dimensional (binormal) 631
  variance of skewness of 606
Gaussian elimination 33f., 51, 55, 1014f.
  fill-in 45, 64
  integral equations 786, 1326
  operation count 34
  outer product variant 1017
  in reduction to Hessenberg form 478, 1231
  relaxation solution of boundary value problems 753ff., 777, 1316
Gaussian function
  Hardy's theorem on Fourier transforms 600
  see also Gaussian (normal) distribution
Gaussian integration 127, 140ff., 789, 1059ff.
  calculation of abscissas and weights 142ff., 1009, 1059ff.
  error estimate in solution 784
  extensions of 153f.
  Golub-Welsch algorithm for weights and abscissas 150, 1064
  for integral equations 781, 783, 1325
  from known recurrence relation 150, 1064

nonclassical weight function 151ff., 788f., 1064f., 1328f.
and orthogonal polynomials 142, 1009, 1061
parallel calculation of formulas 1009, 1061
preassigned nodes 153f.
weight function $\log x$ 153
weight functions 140ff., 788f., 1059ff., 1328f.
Gear's method (stiff ODEs) 730
Geiger counter 266
Generalized eigenvalue problems 455
Generalized minimum residual method (GMRES) 78
Generic interface *see* Interface, generic
Generic procedures 939, 1083, 1094, 1096, 1364
elemental 940, 942, 946f., 1015, 1083
Geometric progression 972, 996f., 1365, 1372ff.
geop() utility function 972, 974, 989, 996, 1127
Geophysics, use of Backus-Gilbert method 809
Gerchberg-Saxton algorithm 805
get_diag() utility function 985, 989, 1005, 1226
Gilbert and Sullivan 714
Givens reduction 462f., 473
fast 463
operation count 463
Glassman, A.J. 180
Global optimization 387f., 436ff., 650, 1219ff.
continuous variables 443f., 1222
Global variables 940, 953f., 1210
allocatable array method 954, 1197, 1212, 1266, 1287, 1298
communicated via internal subprogram 954, 957f., 1067, 1226
danger of 957, 1209, 1293, 1296
pointer method 954, 1197, 1212, 1266, 1287, 1302
Globally convergent
minimization 418ff., 1215
root finding 373, 376ff., 382, 749f., 752, 1196, 1314f.
GMRES (generalized minimum residual method) 78
GNU Emacs 1/xvi
Godunov's method 837
Golden mean (golden ratio) 21, 349, 392f., 399
Golden section search 341, 389ff., 395, 1202ff.
Golub-Welsch algorithm, for Gaussian quadrature 150, 1064
Goodness-of-fit 650, 654, 657f., 662, 690, 1285
GOTO statements, danger of 9, 959
Gram-Schmidt
biorthogonalization 415f.
orthogonalization 94, 450f., 1039
SVD as alternative to 58
Graphics, function plotting 342, 1182f.
Gravitational potential 519
Gray code 300, 881, 886ff., 1344
Greenbaum, A. 79
Gregorian calendar 13, 16, 1011, 1013
Grid square 116f.
Group, dihedral 894, 1345
Guard digits 882, 1343

Half weighting 867, 1337
Halton's quasi-random sequence 300
Hamming window 547
Hamming's motto 341
Hann window 547
Harmonic analysis *see* Fourier transform
Hashing 293, 1144, 1148, 1156
for random number seeds 1147f.
HDLC checksum 890
Heap (data structure) 327f., 336, 897, 1179
Heapsort 320, 327f., 336, 1171f., 1179
Helmholtz equation 852
Hermite polynomials 144, 147
approximation of roots 1062
Hermitian matrix 450ff., 475
Hertz (unit of frequency) 490
Hessenberg matrix 94, 453, 470, 476ff., 488, 1231
see also Matrix
Hessian matrix 382, 408, 415f., 419f., 676ff., 803, 815
is inverse of covariance matrix 667, 679
second derivatives in 676
Hexadecimal constants 17f., 276, 293
initialization 959
Hierarchically band diagonal matrix 598
Hierarchy of program structure 6ff.
High-order not same as high-accuracy 100f., 124, 389, 399, 705, 709, 741
High-pass filter 551
High-Performance Fortran (HPF) 2/xvf., 964, 981, 984
scatter-with-add 1032
Hilbert matrix 83
Home page, Numerical Recipes 1/xx, 2/xvii
Homogeneous linear equations 53
Hook step methods 386
Hotelling's method for matrix inverse 49, 598
Householder transformation 52, 453, 462ff., 469, 473, 475, 478, 481ff., 1227f.
operation count 467
in QR decomposition 92, 1039
HPF *see* High-Performance Fortran
Huffman coding 564, 881, 896f., 902, 1346ff.
huge() intrinsic function 951
Hyperbolic functions, explicit formulas for inverse 178
Hyperbolic partial differential equations 818
advective equation 826
flux-conservative initial value problems 825ff.
Hypergeometric function 202f., 263ff.
routine for 264f., 1138
Hypothesis, null 603

I2B, defined 937

I4B, defined 937
iand() intrinsic function 951
ibclr() intrinsic function 951
ibits() intrinsic function 951
IBM 1/xxiii, 2/xix
  bad random number generator 268
  Fortran 90 compiler 2/viii
  PC 4, 276, 293, 886
  PC-RT 4
  radix base for floating point arithmetic 476
  RS6000 2/viii, 4
IBM checksum 894
ibset() intrinsic function 951
ICCG (incomplete Cholesky conjugate gradient method) 824
ICF (intrinsic correlation function) model 817
Identity (unit) matrix 25
IEEE floating point format 276, 882f., 1343
ieor() intrinsic function 951
if statement, arithmetic 2/xi
if structure 12f.
ifirstloc() utility function 989, 993, 1041, 1346
IIR (infinite impulse response) filter 552ff., 566
Ill-conditioned integral equations 780
Image processing 519, 803
  cosine transform 513
  fast Fourier transform (FFT) 519, 523, 803
  as an inverse problem 803
  maximum entropy method (MEM) 809ff.
  from modulus of Fourier transform 805
  wavelet transform 596f., 1267f.
imaxloc() utility function 989, 993, 1017
iminloc() utility function 989, 993, 1046, 1076
Implicit
  function theorem 340
  pivoting 30, 1014
  shifts in QL method 472ff.
Implicit differencing 827
  for diffusion equation 840
  for stiff equations 729, 740, 1308
IMPLICIT NONE statement 2/xiv, 936
Implied do-list 968, 971, 1127
Importance sampling, in Monte Carlo 306f.
Improper integrals 135ff., 1055
Impulse response function 531, 540, 552
IMSL 1/xxiii, 2/xx, 26, 64, 205, 364, 369, 454
In-place selection 335, 1178f.
Included file, superseded by module 940
Incomplete beta function 219ff., 1098ff.
  for F-test 613, 1271
  routine for 220f., 1097
  for Student's t 610, 613, 1269
Incomplete Cholesky conjugate gradient method (ICCG) 824
Incomplete gamma function 209ff., 1089ff.
  for chi-square 615, 654, 657f., 1272, 1285
  deviates from 282f., 1153
  in mode estimation 610
  routine for 211f., 1089
Increment of linear congruential generator 268
Indentation of blocks 9
Index 934ff., 1446ff.
  this entry 1464
Index loss 967f., 1038
Index table 320, 329f., 1173ff., 1176
Inequality constraints 423
Inheritance 8
Initial value problems 702, 818f.
  see also Differential equations; Partial differential equations
Initialization of derived data type 2/xv
Initialization expression 943, 959, 1012, 1127
Injection operator 864, 1337
Instability *see* Stability
Integer model, in Fortran 90 1144, 1149, 1156
Integer programming 436
Integral equations 779ff.
  adaptive stepsize control 788
  block-by-block method 788
  correspondence with linear algebraic equations 779ff.
  degenerate kernel 785
  eigenvalue problems 780, 785
  error estimate in solution 784
  Fredholm 779f., 782ff., 1325, 1331
  Fredholm alternative 780
  homogeneous, second kind 785, 1325
  ill-conditioned 780
  infinite range 789
  inverse problems 780, 795ff.
  kernel 779
  nonlinear 781, 787
  Nystrom method 782ff., 789, 1325
  product Nystrom method 789, 1328ff.
  with singularities 788ff., 1328ff.
  with singularities, worked example 792, 1328ff.
  subtraction of singularity 789
  symmetric kernel 785
  unstable quadrature 787f.
  Volterra 780f., 786ff., 1326f.
  wavelets 782
  see also Inverse problems
Integral operator, wavelet approximation of 597, 782
Integration of functions 123ff., 1052ff.
  cosine integrals 250, 1125
  Fourier integrals 577ff., 1261
  Fourier integrals, infinite range 583
  Fresnel integrals 248, 1123
  Gauss-Hermite 147f., 1062
  Gauss-Jacobi 148, 1063
  Gauss-Laguerre 146, 1060
  Gauss-Legendre 145, 1059
  integrals that are elliptic integrals 254
  path integration 201ff.
  sine integrals 250, 1125
  see also Quadrature
Integro-differential equations 782
INTENT attribute 1072, 1092
Interface (Fortran 90) 939, 942, 1067

for communication between program parts 957, 1209, 1293, 1296
explicit 939, 942, 1067, 1384
generic 2/xiii, 940, 1015, 1083, 1094, 1096
implicit 939
for Numerical Recipes 1384ff.
Interface block 939, 1084, 1384
Interface, in programs 2, 8
Intermediate value theorem 343
Internal subprogram (Fortran 90) 2/xiv, 954, 957, 1067, 1202f., 1226
nesting of 2/xii
resembles C macro 1302
supersedes statement function 1057, 1256
International Standards Organization (ISO) 2/xf., 2/xiii
Internet, availability of code over 1/xx, 2/xvii
Interpolation 99ff.
Aitken's algorithm 102
avoid 2-stage method 100
avoid in Fourier analysis 569
bicubic 118f., 1049f.
bilinear 117
caution on high-order 100
coefficients of polynomial 100, 113ff., 191, 575, 1047f., 1078
for computing Fourier integrals 578
error estimates for 100
of functions with poles 104ff., 1043f.
inverse quadratic 353, 395ff., 1204
multidimensional 101f., 116ff., 1049ff.
in multigrid method 866, 1337
Neville's algorithm 102f., 182, 1043
Nystrom 783, 1326
offset arrays 104, 113
operation count for 100
operator 864, 1337
order of 100
and ordinary differential equations 101
oscillations of polynomial 100, 116, 389, 399
parabolic, for minimum finding 395, 1204
polynomial 99, 102ff., 182, 1043
rational Chebyshev approximation 197ff., 1081
rational function 99, 104ff., 194ff., 225, 718ff., 726, 1043f., 1080, 1306
reverse (extirpolation) 574, 1261
spline 100, 107ff., 120f., 1044f., 1050f.
trigonometric 99
see also Fitting
Interprocessor communication 969, 981
Interval variable (statistics) 623
Intrinsic correlation function (ICF) model 817
Intrinsic data types 937
Intrinsic procedures
array inquiry 938, 942, 948ff.
array manipulation 950
array reduction 948
array unary and binary functions 949
backwards-compatibility 946
bit manipulation 2/xiii, 951
character 952
cmplx 1254
conversion elemental 946
elemental 940, 942, 946f., 951, 1083, 1364
generic 939, 1083f., 1364
lexical comparison 952
numeric inquiry 2/xiv, 1107, 1231, 1343
numerical 946, 951f.
numerical representation 951
pack used for sorting 1171
random_number 1143
real 1254
top 10 945
truncation 946f.
see also Fortran 90
Inverse hyperbolic function 178, 255
Inverse iteration *see* Eigensystems
Inverse problems 779, 795ff.
Backus-Gilbert method 806ff.
Bayesian approach 799, 810f., 816f.
central idea 799
constrained linear inversion method 799ff.
data inversion 807
deterministic constraints 804ff.
in geophysics 809
Gerchberg-Saxton algorithm 805
incomplete Fourier coefficients 813
and integral equations 780
linear regularization 799ff.
maximum entropy method (MEM) 810, 815f.
MEM demystified 814
Phillips-Twomey method 799ff.
principal solution 797
regularization 796ff.
regularizing operator 798
stabilizing functional 798
Tikhonov-Miller regularization 799ff.
trade-off curve 795
trade-off curve, Backus-Gilbert method 809
two-dimensional regularization 803
use of conjugate gradient minimization 804, 815
use of convex sets 804
use of Fourier transform 803, 805
Van Cittert's method 804
Inverse quadratic interpolation 353, 395ff., 1204
Inverse response kernel, in Backus-Gilbert method 807
Inverse trigonometric function 255
ior() intrinsic function 951
ISBN (International Standard Book Number) checksum 894
ishft() intrinsic function 951
ishftc() intrinsic function 951
ISO (International Standards Organization) 2/xf., 2/xiii
Iterated integrals 155
Iteration 9f.
functional 740f.
to improve solution of linear algebraic equations 47ff., 195, 1022
for linear algebraic equations 26

required for two-point boundary value problems 745
in root finding 340f.
Iteration matrix 856
ITPACK 71
Iverson, John 2/xi

Jacobi matrix, for Gaussian quadrature 150, 1064
Jacobi polynomials, approximation of roots 1064
Jacobi transformation (or rotation) 94, 453, 456ff., 462, 475, 489, 1041, 1225
Jacobian determinant 279, 774
Jacobian elliptic functions 261, 1137f.
Jacobian matrix 374, 376, 379, 382, 731, 1197f., 1309
singular in Newton's rule 386
Jacobi's method (relaxation) 855ff., 864
Jenkins-Traub method 369
Julian Day 1, 13, 16, 936, 1010ff.
Jump transposition errors 895

K-S test *see* Kolmogorov-Smirnov test
Kalman filter 700
Kanji 2/xii
Kaps-Rentrop method 730, 1308
Kendall's tau 634, 637ff., 1279
Kennedy, Ken 2/xv
Kepler's equation 1061
Kermit checksum 889
Kernel 779
averaging, in Backus-Gilbert method 807
degenerate 785
finite rank 785
inverse response 807
separable 785
singular 788f., 1328
symmetric 785
Keys used in sorting 329, 889
Keyword argument 2/xiv, 947f., 1341
kind() intrinsic function 951
KIND parameter 946, 1261, 1284
and cmplx() intrinsic function 1125, 1192, 1254
default 937
for Numerical Recipes 1361
for random numbers 1144
and real() intrinsic function 1125
Kolmogorov-Smirnov test 614, 617ff., 694, 1273f.
two-dimensional 640, 1281ff.
variants 620ff., 640, 1281
Kuiper's statistic 621
Kurtosis 606, 608, 1269

L-estimate 694
Labels, statement 9
Lag 492, 538, 553
Lagged Fibonacci generator 1142, 1148ff.
Lagrange multiplier 795
Lagrange's formula for polynomial interpolation 84, 102f., 575, 578
Laguerre polynomials, approximation of roots 1061
Laguerre's method 341, 365f., 1191f.
Lanczos lemma 498f.
Lanczos method for gamma function 206, 1085
Landen transformation 256
LAPACK 26, 1230
Laplace's equation 246, 818
see also Poisson equation
Las Vegas 625
Latin square or hypercube 305f.
Laurent series 566
Lax method 828ff., 836, 845f.
multidimensional 845f.
Lax-Wendroff method 835ff.
lbound() intrinsic function 949
Leakage in power spectrum estimation 544, 548
Leakage width 548f.
Leapfrog method 833f.
Least squares filters *see* Savitzky-Golay filters
Least squares fitting 645, 651ff., 655ff., 660ff., 665ff., 1285f., 1288f.
contrasted to general minimization problems 684ff.
degeneracies in 671f., 674
Fourier components 570
as M-estimate for normal errors 696
as maximum likelihood estimator 652
as method for smoothing data 645, 1283
Fourier components 1258
freezing parameters in 668, 700
general linear case 665ff., 1288, 1290f.
Levenberg-Marquardt method 678ff., 816, 1292f.
Lomb periodogram 570, 1258
multidimensional 675
nonlinear 386, 675ff., 816, 1292
nonlinear, advanced methods 683
normal equations 645, 666f., 800, 1288
normal equations often singular 670, 674
optimal (Wiener) filtering 540f.
QR method in 94, 668
for rational Chebyshev approximation 199f., 1081f.
relation to linear correlation 630, 658
Savitzky-Golay filter as 645, 1283
singular value decomposition (SVD) 25f., 51ff., 199f., 670ff., 1081, 1290
skewed by outliers 653
for spectral analysis 570, 1258
standard (probable) errors on fitted parameters 667, 671
weighted 652
see also Fitting
L'Ecuyer's long period random generator 271, 273
Least squares fitting
standard (probable) errors on fitted parameters 1288, 1290
weighted 1285
Left eigenvalues or eigenvectors 451
Legal matters 1/xx, 2/xvii
Legendre elliptic integral *see* Elliptic integrals

Legendre polynomials 246, 1122
  fitting data to 674, 1291f.
  recurrence relation 172
  shifted monic 151
  see also Associated Legendre polynomials; Spherical harmonics
Lehmer-Schur algorithm 369
Lemarie's wavelet 593
Lentz's method for continued fraction 165, 212
Lepage, P. 309
Leptokurtic distribution 606
Levenberg-Marquardt algorithm 386, 678ff., 816, 1292
  advanced implementation 683
Levinson's method 86, 1038
Lewis, H.W. 275
Lexical comparison functions 952
LGT, defined 937
License information 1/xx, 2/xviiff.
Limbo 356
Limit cycle, in Laguerre's method 365
Line minimization *see* Minimization, along a ray
Line search *see* Minimization, along a ray
Linear algebra, intrinsic functions for parallelization 969f., 1026, 1040, 1200, 1326
Linear algebraic equations 22ff., 1014
  band diagonal 43ff., 1019
  biconjugate gradient method 77, 1034ff.
  Cholesky decomposition 89f., 423, 455, 668, 1038f.
  complex 41
  computing $\mathbf{A}^{-1} \cdot \mathbf{B}$ 40
  conjugate gradient method 77ff., 599, 1034
  cyclic tridiagonal 67, 1030
  direct methods 26, 64, 1014, 1030
  Fortran 90 vs. library routines 1016
  Gauss-Jordan elimination 27ff., 1014
  Gaussian elimination 33f., 1014f.
  Hilbert matrix 83
  Hotelling's method 49, 598
  and integral equations 779ff., 783, 1325
  iterative improvement 47ff., 195, 1022
  iterative methods 26, 77ff., 1034
  large sets of 23
  least squares solution 53ff., 57f., 199f., 671, 1081, 1290
  LU decomposition 34ff., 195, 386, 732, 783, 786, 801, 1016, 1022, 1325f.
  nonsingular 23
  overdetermined 25f., 199, 670, 797
  partitioned 70
  QR decomposition 91f., 382, 386, 668, 1039f., 1199
  row vs. column elimination 31f.
  Schultz's method 49, 598
  Sherman-Morrison formula 65ff., 83
  singular 22, 53, 58, 199, 670
  singular value decomposition (SVD) 51ff., 199f., 670ff., 797, 1022, 1081, 1290
  sparse 23, 43, 63ff., 732, 804, 1020f., 1030
  summary of tasks 25f.
  Toeplitz 82, 85ff., 195, 1038
  tridiagonal 26, 42f., 64, 109, 150, 453f., 462ff., 469ff., 488, 839f., 853, 861f., 1018f., 1227ff.
  Vandermonde 82ff., 114, 1037, 1047
  wavelet solution 597ff., 782
  Woodbury formula 68ff., 83
  see also Eigensystems
Linear congruential random number generator 267ff., 1142
  choice of constants for 274ff.
Linear constraints 423
Linear convergence 346, 393
Linear correlation (statistics) 630ff., 1276
Linear dependency
  constructing orthonormal basis 58, 94
  of directions in N-dimensional space 409
  in linear algebraic equations 22f.
Linear equations *see* Differential equations; Integral equations; Linear algebraic equations
Linear inversion method, constrained 799ff.
Linear prediction 557ff.
  characteristic polynomial 559
  coefficients 557ff., 1256
  compared to maximum entropy method 558
  compared with regularization 801
  contrasted to polynomial extrapolation 560
  related to optimal filtering 558
  removal of bias in 563
  stability 559f., 1257
Linear predictive coding (LPC) 563ff.
Linear programming 387, 423ff., 1216ff.
  artificial variables 429
  auxiliary objective function 430
  basic variables 426
  composite simplex algorithm 435
  constraints 423
  convergence criteria 432
  degenerate feasible vector 429
  dual problem 435
  equality constraints 423
  feasible basis vector 426
  feasible vector 424
  fundamental theorem 426
  inequality constraints 423
  left-hand variables 426
  nonbasic variables 426
  normal form 426
  objective function 424
  optimal feasible vector 424
  pivot element 428f.
  primal-dual algorithm 435
  primal problem 435
  reduction to normal form 429ff.
  restricted normal form 426ff.
  revised simplex method 435
  right-hand variables 426
  simplex method 402, 423ff., 431ff., 1216ff.
  slack variables 429
  tableau 427
  vertex of simplex 426

Linear recurrence *see* Recurrence relation
Linear regression 655ff., 660ff., 1285ff.
see also Fitting
Linear regularization 799ff.
LINPACK 26
Literal constant 937, 1361
Little-endian 293
Local extrapolation 709
Local extremum 387f., 437
Localization of roots *see* Bracketing
Logarithmic function 255
Lomb periodogram method of spectral analysis 569f., 1258f.
fast algorithm 574f., 1259
Loops 9f.
Lorentzian probability distribution 282, 696f.
Low-pass filter 551, 644f., 1283f.
Lower subscript 944
lower_triangle() utility function 989, 1007, 1200
LP coefficients *see* Linear prediction
LPC (linear predictive coding) 563ff.
LU decomposition 34ff., 47f., 51, 55, 64, 97, 374, 667, 732, 1016, 1022
for $\mathbf{A}^{-1} \cdot \mathbf{B}$ 40
backsubstitution 39, 1017
band diagonal matrix 43ff., 1020
complex equations 41f.
Crout's algorithm 36ff., 45, 1017
for integral equations 783, 786, 1325f.
for inverse iteration of eigenvectors 488
for inverse problems 801
for matrix determinant 41
for matrix inverse 40, 1016
for nonlinear sets of equations 374, 386, 1196
operation count 36, 39
outer product Gaussian elimination 1017
for Padé approximant 195, 1080
pivoting 37f., 1017
repeated backsubstitution 40, 46
solution of linear algebraic equations 40, 1017
solution of normal equations 667
for Toeplitz matrix 87
Lucifer 290

M&R (Metcalf and Reid) 935
M-estimates 694ff.
how to compute 697f.
local 695ff.
see also Maximum likelihood estimate
Machine accuracy 19f., 881f., 1189, 1343
Macintosh, *see* Apple Macintosh
Maehly's procedure 364, 371
Magic
in MEM image restoration 814
in Padé approximation 195
Mantissa in floating point format 19, 882, 909, 1343
Marginals 624
Marquardt method (least squares fitting) 678ff., 816, 1292f.
Marsaglia shift register 1142, 1148ff.
Marsaglia, G. 1142, 1149
mask 1006f., 1102, 1200, 1226, 1305, 1333f., 1368, 1378, 1382
optional argument 948
optional argument, facilitates parallelism 967f., 1038
Mass, center of 295ff.
MasterCard checksum 894
Mathematical Center (Amsterdam) 353
Mathematical intrinsic functions 946, 951f.
matmul() intrinsic function 945, 949, 969, 1026, 1040, 1050, 1076, 1200, 1216, 1290, 1326
Matrix 23ff.
add vector to diagonal 1004, 1234, 1366, 1381
approximation of 58f., 598f.
band diagonal 42ff., 64, 1019
band triangular 64
banded 26, 454
bidiagonal 52
block diagonal 64, 754
block triangular 64
block tridiagonal 64
bordered 64
characteristic polynomial 449, 469
Cholesky decomposition 89f., 423, 455, 668, 1038f.
column augmented 28, 1014
complex 41
condition number 53, 78
create unit matrix 1006, 1382
curvature 677
cyclic banded 64
cyclic tridiagonal 67, 1030
defective 450, 476, 489
of derivatives *see* Hessian matrix; Jacobian determinant
design (fitting) 645, 665, 801, 1082
determinant of 25, 41
diagonal of sparse matrix 1033ff.
diagonalization 452ff., 1225ff.
elementary row and column operations 28f.
finite differencing of partial differential equations 821ff.
get diagonal 985, 1005, 1226f., 1366, 1381f.
Hermitian 450, 454, 475
Hermitian conjugate 450
Hessenberg 94, 453, 470, 476ff., 488, 1231ff.
Hessian *see* Hessian matrix
hierarchically band diagonal 598
Hilbert 83
identity 25
ill-conditioned 53, 56, 114
indexed storage of 71f., 1030
and integral equations 779, 783, 1325
inverse 25, 27, 34, 40, 65ff., 70, 95ff., 1014, 1016f.
inverse, approximate 49
inverse by Hotelling's method 49, 598
inverse by Schultz's method 49, 598
inverse multiplied by a matrix 40
iteration for inverse 49, 598

Jacobi transformation 453, 456ff., 462, 1225f.
Jacobian 731, 1309
logical dimension 24
lower triangular 34f., 89, 781, 1016
lower triangular mask 1007, 1200, 1382
multiplication denoted by dot 23
multiplication, intrinsic function 949, 969, 1026, 1040, 1050, 1200, 1326
norm 50
normal 450ff.
nullity 53
nullspace 25, 53f., 449, 795
orthogonal 91, 450, 463ff., 587
orthogonal transformation 452, 463ff., 469, 1227
orthonormal basis 58, 94
outer product denoted by cross 66, 420
partitioning for determinant 70
partitioning for inverse 70
pattern multiply of sparse 74
physical dimension 24
positive definite 26, 89f., 668, 1038
QR decomposition 91f., 382, 386, 668, 1039, 1199
range 53
rank 53
residual 49
row and column indices 23
row vs. column operations 31f.
self-adjoint 450
set diagonal elements 1005, 1200, 1366, 1382
similarity transform 452ff., 456, 476, 478, 482
singular 53f., 58, 449
singular value decomposition 26, 51ff., 797
sparse 23, 63ff., 71, 598, 732, 754, 804, 1030ff.
special forms 26
splitting in relaxation method 856f.
spread 808
square root of 423, 455
symmetric 26, 89, 450, 454, 462ff., 668, 785, 1038, 1225, 1227
threshold multiply of sparse 74, 1031
Toeplitz 82, 85ff., 195, 1038
transpose() intrinsic function 950
transpose of sparse 73f., 1033
triangular 453
tridiagonal 26, 42f., 64, 109, 150, 453f., 462ff., 469ff., 488, 839f., 853, 861f., 1018f., 1227ff.
tridiagonal with fringes 822
unitary 450
updating 94, 382, 386, 1041, 1199
upper triangular 34f., 91, 1016
upper triangular mask 1006, 1226, 1305, 1382
Vandermonde 82ff., 114, 1037, 1047
see also Eigensystems
Matrix equations *see* Linear algebraic equations
Matterhorn 606
maxexponent() intrinsic function 1107
Maximization *see* Minimization
Maximum entropy method (MEM) 565ff., 1258
algorithms for image restoration 815f.
Bayesian 816f.
Cornwell-Evans algorithm 816
demystified 814
historic vs. Bayesian 816f.
image restoration 809ff.
intrinsic correlation function (ICF) model 817
for inverse problems 809ff.
operation count 567
see also Linear prediction
Maximum likelihood estimate (M-estimates) 690, 694ff.
and Bayes' Theorem 811
chi-square test 690
defined 652
how to compute 697f.
mean absolute deviation 696, 698, 1294
relation to least squares 652
maxloc() intrinsic function 949, 992f., 1015
modified in Fortran 95 961
maxval() intrinsic function 945, 948, 961, 1016, 1273
Maxwell's equations 825f.
Mean(s)
of distribution 604f., 608f., 1269
statistical differences between two 609ff., 1269f.
Mean absolute deviation of distribution 605, 696, 1294
related to median 698
Measurement errors 650
Median 320
calculating 333
of distribution 605, 608f.
as L-estimate 694
role in robust straight line fitting 698
by selection 698, 1294
Median-of-three, in Quicksort 324
MEM *see* Maximum entropy method (MEM)
Memory leak 953, 956, 1071, 1327
Memory management 938, 941f., 953ff., 1327, 1336
merge construct 945, 950, 1099f.
for conditional scalar expression 1010, 1094f.
contrasted with where 1023
parallelization 1011
Merge-with-dummy-values idiom 1090
Merit function 650
in general linear least squares 665
for inverse problems 797
nonlinear models 675
for straight line fitting 656, 698
for straight line fitting, errors in both coordinates 660, 1286
Mesh-drift instability 834f.
Mesokurtic distribution 606
Metcalf, Michael 2/viii
see also M&R
Method of regularization 799ff.

Metropolis algorithm 437f., 1219
Microsoft 1/xxii, 2/xix
Microsoft Fortran PowerStation 2/viii
Midpoint method *see* Modified midpoint method; Semi-implicit midpoint rule
Mikado, or Town of Titipu 714
Miller's algorithm 175, 228, 1106
MIMD machines (Multiple Instruction Multiple Data) 964, 985, 1071, 1084
Minimal solution of recurrence relation 174
Minimax polynomial 186, 198, 1076
Minimax rational function 198
Minimization 387ff.
  along a ray 77, 376f., 389, 406ff., 412f., 415f., 418, 1195f., 1211, 1213
  annealing, method of simulated 387f., 436ff., 1219ff.
  bracketing of minimum 390ff., 402, 1201f.
  Brent's method 389, 395ff., 399, 660f., 1204ff., 1286
  Broyden-Fletcher-Goldfarb-Shanno algorithm 390, 418ff., 1215
  chi-square 653ff., 675ff., 1285, 1292
  choice of methods 388f.
  combinatorial 436f., 1219
  conjugate gradient method 390, 413ff., 804, 815, 1210, 1214
  convergence rate 393, 409
  Davidon-Fletcher-Powell algorithm 390, 418ff., 1215
  degenerate 795
  direction-set methods 389, 406ff., 1210ff.
  downhill simplex method 389, 402ff., 444, 697f., 1208, 1222ff.
  finding best-fit parameters 650
  Fletcher-Reeves algorithm 390, 414ff., 1214
  functional 795
  global 387f., 443f., 650, 1219, 1222
  globally convergent multidimensional 418, 1215
  golden section search 390ff., 395, 1202ff.
  multidimensional 388f., 402ff., 1208ff., 1214
  in nonlinear model fitting 675f., 1292
  Polak-Ribiere algorithm 389, 414ff., 1214
  Powell's method 389, 402, 406ff., 1210ff.
  quasi-Newton methods 376, 390, 418ff., 1215
  and root finding 375
  scaling of variables 420
  by searching smaller subspaces 815
  steepest descent method 414, 804
  termination criterion 392, 404
  use in finding double roots 341
  use for sparse linear systems 77ff.
  using derivatives 389f., 399ff., 1205ff.
  variable metric methods 390, 418ff., 1215
  see also Linear programming
Minimum residual method, for sparse system 78
minloc() intrinsic function 949, 992f.
  modified in Fortran 95 961
MINPACK 683
minval() intrinsic function 948, 961
MIPS 886
Missing data problem 569
Mississippi River 438f., 447
MMP (massively multiprocessor) machines 965ff., 974, 981, 984, 1016ff., 1021, 1045, 1226ff., 1250
Mode of distribution 605, 609
Modeling of data *see* Fitting
Model-trust region 386, 683
Modes, homogeneous, of recursive filters 554
Modified Bessel functions *see* Bessel functions
Modified Lentz's method, for continued fractions 165
Modified midpoint method 716ff., 720, 1302f.
Modified moments 152
Modula-2 7
Modular arithmetic, without overflow 269, 271, 275
Modular programming 2/xiii, 7f., 956ff., 1209, 1293, 1296, 1346
MODULE facility 2/xiii, 936f., 939f., 957, 1067, 1298, 1320, 1322, 1324, 1330, 1346
  initializing random number generator 1144ff.
  in nr.f90 936, 941f., 1362, 1384ff.
  in nrtype.f90 936f., 1361f.
  in nrutil.f90 936, 1070, 1362, 1364ff.
  sparse matrix 1031
  undefined variables on exit 953, 1266
Module subprogram 940
modulo() intrinsic function 946, 1156
Modulus of linear congruential generator 268
Moments
  of distribution 604ff., 1269
  filter that preserves 645
  modified problem of 151f.
  problem of 83
  and quadrature formulas 791, 1328
  semi-invariants 608
Monic polynomial 142f.
Monotonicity constraint, in upwind differencing 837
Monte Carlo 155ff., 267
  adaptive 306ff., 1161ff.
  bootstrap method 686f.
  comparison of sampling methods 309
  exploration of binary tree 290
  importance sampling 306f.
  integration 124, 155ff., 295ff., 306ff., 1161
  integration, recursive 314ff., 1164ff.
  integration, using Sobol' sequence 304
  integration, VEGAS algorithm 309ff., 1161
  and Kolmogorov-Smirnov statistic 622, 640
  partial differential equations 824
  quasi-random sequences in 299ff.
  quick and dirty 686f.
  recursive 306ff., 314ff., 1161, 1164ff.
  significance of Lomb periodogram 570
  simulation of data 654, 684ff., 690
  stratified sampling 308f., 314, 1164

Moon, calculate phases of 1f., 14f., 936, 1010f.
Mother functions 584
Mother Nature 684, 686
Moving average (MA) model 566
Moving window averaging 644
Mozart 9
MS 1/xxii, 2/xix
Muller's method 364, 372
Multidimensional
  confidence levels of fitting 688f.
  data, use of binning 623
  Fourier transform 515ff., 1241, 1246, 1251
  Fourier transform, real data 519ff., 1248f.
  initial value problems 844ff.
  integrals 124, 155ff., 295ff., 306ff., 1065ff., 1161ff.
  interpolation 116ff., 1049ff.
  Kolmogorov-Smirnov test 640, 1281
  least squares fitting 675
  minimization 402ff., 406ff., 413ff., 1208ff., 1214f., 1222ff.
  Monte Carlo integration 295ff., 306ff., 1161ff.
  normal (Gaussian) distribution 690
  optimization 388f.
  partial differential equations 844ff.
  root finding 340ff., 358, 370, 372ff., 746, 749f., 752, 754, 1194ff., 1314ff.
  search using quasi-random sequence 300
  secant method 373, 382f., 1199f.
  wavelet transform 595, 1267f.
Multigrid method 824, 862ff., 1334ff.
  avoid SOR 866
  boundary conditions 868f.
  choice of operators 868
  coarse-to-fine operator 864, 1337
  coarse-grid correction 864f.
  cycle 865
  dual viewpoint 875
  fine-to-coarse operator 864, 1337
  full approximation storage (FAS) algorithm 874, 1339ff.
  full multigrid method (FMG) 863, 868, 1334ff.
  full weighting 867
  Gauss-Seidel relaxation 865f., 1338
  half weighting 867, 1337
  importance of adjoint operator 867
  injection operator 864, 1337
  interpolation operator 864, 1337
  line relaxation 866
  local truncation error 875
  Newton's rule 874, 876, 1339, 1341
  nonlinear equations 874ff., 1339ff.
  nonlinear Gauss-Seidel relaxation 876, 1341
  odd-even ordering 866, 869, 1338
  operation count 862
  prolongation operator 864, 1337
  recursive nature 865, 1009, 1336
  relative truncation error 875
  relaxation as smoothing operator 865
  restriction operator 864, 1337
  speeding up FMG algorithm 873
  stopping criterion 875f.
  straight injection 867
  symbol of operator 866f.
  use of Richardson extrapolation 869
  V-cycle 865, 1336
  W-cycle 865, 1336
  zebra relaxation 866
Multiple precision arithmetic 906ff., 1352ff.
Multiple roots 341, 362
Multiplication, complex 171
Multiplication, multiple precision 907, 909, 1353f.
Multiplier of linear congruential generator 268
Multistep and multivalue methods (ODEs) 740ff.
  see also Differential Equations; Predictor-corrector methods
Multivariate normal distribution 690
Murphy's Law 407
Musical scores 5f.

NAG 1/xxiii, 2/xx, 26, 64, 205, 454
  Fortran 90 compiler 2/viii, 2/xiv
Named constant 940
  initialization 1012
  for Numerical Recipes 1361
Named control structure 959, 1219, 1305
National Science Foundation (U.S.) 1/xvii, 1/xix, 2/ix
Natural cubic spline 109, 1044f.
Navier-Stokes equation 830f.
nearest() intrinsic function 952, 1146
Needle, eye of (minimization) 403
Negation, multiple precision 907, 1353f.
Negentropy 811, 896
Nelder-Mead minimization method 389, 402, 1208
Nested iteration 868
Neumann boundary conditions 820, 840, 851, 858
Neutrino 640
Neville's algorithm 102f., 105, 134, 182, 1043
Newton-Cotes formulas 125ff., 140
Newton-Raphson method *see* Newton's rule
Newton's rule 143f., 180, 341, 355ff., 362, 364, 469, 1059, 1189
  with backtracking 376, 1196
  caution on use of numerical derivatives 356ff.
  fractal domain of convergence 360f.
  globally convergent multidimensional 373, 376ff., 382, 749f., 752, 1196, 1199, 1314f.
  for matrix inverse 49, 598
  in multidimensions 370, 372ff., 749f., 752, 754, 1194ff., 1314ff.
  in nonlinear multigrid 874, 876, 1339, 1341
  nonlinear Volterra equations 787
  for reciprocal of number 911, 1355
  safe 359, 1190
  scaling of variables 381

singular Jacobian 386
solving stiff ODEs 740
for square root of number 912, 1356
Niederreiter sequence 300
NL2SOL 683
Noise
bursty 889
effect on maximum entropy method 567
equivalent bandwidth 548
fitting data which contains 647f., 650
model, for optimal filtering 541
Nominal variable (statistics) 623
Nonexpansive projection operator 805
Non-interfering directions *see* Conjugate directions
Nonlinear eigenvalue problems 455
Nonlinear elliptic equations, multigrid method 874ff., 1339ff.
Nonlinear equations, in MEM inverse problems 813
Nonlinear equations, roots of 340ff.
Nonlinear instability 831
Nonlinear integral equations 781, 787
Nonlinear programming 436
Nonnegativity constraints 423
Nonparametric statistics 633ff., 1277ff.
Nonpolynomial complete (NP-complete) 438
Norm, of matrix 50
Normal (Gaussian) distribution 267, 652, 682, 798, 1294
central limit theorem 652f.
deviates from 279f., 571, 1152
kurtosis of 607
multivariate 690
semi-invariants of 608
tails compared to Poisson 653
two-dimensional (binormal) 631
variance of skewness of 606
Normal equations (fitting) 26, 645, 666ff., 795, 800, 1288
often are singular 670
Normalization
of Bessel functions 175
of floating-point representation 19, 882, 1343
of functions 142, 765
of modified Bessel functions 232
not() intrinsic function 951
Notch filter 551, 555f.
NP-complete problem 438
nr.f90 (module file) 936, 1362, 1384ff.
nrerror() utility function 989, 995
nrtype.f90 (module file) 936f.
named constants 1361
nrutil.f90 (module file) 936, 1070, 1362, 1364ff.
table of contents 1364
Null hypothesis 603
nullify statement 953f., 1070, 1302
Nullity 53
Nullspace 25, 53f., 449, 795
Number-theoretic transforms 503f.
Numeric inquiry functions 2/xiv, 1107, 1231, 1343
Numerical derivatives 180ff., 645, 1075
Numerical integration *see* Quadrature
Numerical intrinsic functions 946, 951f.
Numerical Recipes
compatibility with First Edition 4
Example Book 3
Fortran 90 types 936f., 1361
how to get programs 1/xx, 2/xvii
how to report bugs 1/iv, 2/iv
interface blocks (Fortran 90) 937, 941f., 1084, 1384ff.
no warranty on 1/xx, 2/xvii
plan of two-volume edition 1/xiii
table of dependencies 921ff., 1434ff.
as trademark 1/xxiii, 2/xx
utility functions (Fortran 90) 936f., 945, 968, 970, 972ff., 977, 984, 987ff., 1015, 1071f., 1361ff.
Numerical Recipes Software 1/xv, 1/xxiiff., 2/xviiff.
address and fax number 1/iv, 1/xxii, 2/iv, 2/xix
Web home page 1/xx, 2/xvii
Nyquist frequency 494ff., 520, 543, 545, 569ff.
Nystrom method 782f., 789, 1325
product version 789, 1331

Object extensibility 8
Objective function 424
Object-oriented programming 2/xvi, 2, 8
Oblateness parameter 764
Obsolete features *see* Fortran, Obsolescent features
Octal constant, initialization 959
Odd-even ordering
allows parallelization 1333
in Gauss-Seidel relaxation 866, 869, 1338
in successive over-relaxation (SOR) 859, 1332
Odd parity 888
OEM information 1/xxii
One-sided power spectral density 492
ONLY option, for USE statement 941, 957, 1067
Operation count
balancing 476
Bessel function evaluation 228
bisection method 346
Cholesky decomposition 90
coefficients of interpolating polynomial 114f.
complex multiplication 97
cubic spline interpolation 109
evaluating polynomial 168
fast Fourier transform (FFT) 498
Gauss-Jordan elimination 34, 39
Gaussian elimination 34
Givens reduction 463
Householder reduction 467
interpolation 100
inverse iteration 488
iterative improvement 48
Jacobi transformation 460
Kendall's tau 637

linear congruential generator 268
LU decomposition 36, 39
matrix inversion 97
matrix multiplication 96
maximum entropy method 567
multidimensional minimization 413f.
multigrid method 862
multiplication 909
polynomial evaluation 97f., 168
QL method 470, 473
QR decomposition 92
QR method for Hessenberg matrices 484
reduction to Hessenberg form 479
selection by partitioning 333
sorting 320ff.
Spearman rank-order coefficient 638
Toeplitz matrix 83
Vandermonde matrix 83
Operator overloading 2/xiif., 7
Operator splitting 823, 847f., 861
Operator, user-defined 2/xii
Optimal feasible vector 424
Optimal (Wiener) filtering 535, 539ff., 558, 644
compared with regularization 801
Optimization *see* Minimization
Optimization of code 2/xiii
Optional argument 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
dim 948
mask 948, 968, 1038
testing for 952
Ordering Numerical Recipes 1/xxf., 2/xviif.
Ordinal variable (statistics) 623
Ordinary differential equations *see* Differential equations
Orthogonal *see* Orthonormal functions; Orthonormal polynomials
Orthogonal transformation 452, 463ff., 469, 584, 1227
Orthonormal basis, constructing 58, 94, 1039
Orthonormal functions 142, 246
Orthonormal polynomials
Chebyshev 144, 184ff., 1076ff.
construct for arbitrary weight 151ff., 1064
in Gauss-Hermite integration 147, 1062
and Gaussian quadrature 142, 1009, 1061
Gaussian weights from recurrence 150, 1064
Hermite 144, 1062
Jacobi 144, 1063
Laguerre 144, 1060
Legendre 144, 1059
weight function $\log x$ 153
Orthonormality 51, 142, 463
Outer product Gaussian elimination 1017
Outer product of matrices (denoted by cross) 66, 420, 949, 969f., 989, 1000ff., 1017, 1026, 1040, 1076, 1200, 1216, 1275
outerand() utility function 989, 1002, 1015
outerdiff() utility function 989, 1001
outerdiv() utility function 989, 1001
outerprod() utility function 970, 989, 1000, 1017, 1026, 1040, 1076, 1200, 1216, 1275
outersum() utility function 989, 1001
Outgoing wave boundary conditions 820
Outlier 605, 653, 656, 694, 697
see also Robust estimation
Overcorrection 857
Overflow 882, 1343
how to avoid in modulo multiplication 269
in complex arithmetic 171
Overlap-add and overlap-save methods 536f.
Overloading
operator 2/xiif.
procedures 940, 1015, 1083, 1094, 1096
Overrelaxation parameter 857, 1332
choice of 858

Pack() intrinsic function 945, 950, 964, 991, 1031
communication bottleneck 969
for index table 1176
for partition-exchange 1170
for selection 1178
for selective evaluation 1087
Pack-unpack idiom 1087, 1134, 1153
Padé approximant 194ff., 1080f.
Padé approximation 105
Parabolic interpolation 395, 1204
Parabolic partial differential equations 818, 838ff.
Parallel axis theorem 308
Parallel programming 2/xv, 941, 958ff., 962ff., 965f., 968f., 987
array operations 964f.
array ranking 1278f.
band diagonal linear equations 1021
Bessel functions 1107ff.
broadcasts 965ff.
C and C++ 2/viii
communication costs 969, 981, 1250
counting do-loops 1015
cyclic reduction 974
deflation 977ff.
design matrix 1082
dimensional expansion 965ff.
eigensystems 1226, 1229f.
fast Fourier transform (FFT) 981, 1235ff., 1250
in Fortran 90 963ff.
Fortran 90 tricks 1009, 1274, 1278, 1280
function evaluation 986, 1009, 1084f., 1087, 1090, 1102, 1128, 1134
Gaussian quadrature 1009, 1061
geometric progressions 972
index loss 967f., 1038
index table 1176f.
interprocessor communication 981
Kendall's tau 1280
linear algebra 969f., 1000ff., 1018f., 1026, 1040, 1200, 1326
linear recurrence 973f., 1073ff.
logo 2/viii, 1009
masks 967f., 1006f., 1038, 1102, 1200, 1226, 1305, 1333f., 1368, 1378, 1382
merge statement 1010

MIMD (multiple instruction, multiple data) 964, 985f., 1084
MMP (massively multiprocessor) machines 965ff., 974, 984, 1016ff., 1226ff., 1250
nrutil.f90 (module file) 1364ff.
odd-even ordering 1333
one-dimensional FFT 982f.
parallel note icon 1009
partial differential equations 1333
in-place selection 1178f.
polynomial coefficients from roots 980
polynomial evaluation 972f., 977, 998
random numbers 1009, 1141ff.
recursive doubling 973f., 976f., 979, 988, 999, 1071ff.
scatter-with-combine 984, 1002f., 1032f.
second order recurrence 974f., 1074
SIMD (Single Instruction Multiple Data) 964, 985f., 1009, 1084f.
singular value decomposition (SVD) 1026
sorting 1167ff., 1171, 1176f.
special functions 1009
SSP (small-scale parallel) machines 965ff., 984, 1010ff., 1016ff., 1059f., 1226ff., 1250
subvector scaling 972, 974, 996, 1000
successive over-relaxation (SOR) 1333
supercomputers 2/viii, 962
SVD algorithm 1026
synthetic division 977ff., 999, 1048, 1071f., 1079, 1192
tridiagonal systems 975f., 1018, 1229f.
utilities 1364ff.
vector reduction 972f., 977, 998
vs. serial programming 965, 987
PARAMETER attribute 1012
Parameters in fitting function 651, 684ff.
Parity bit 888
Park and Miller minimal standard random generator 269, 1142
Parkinson's Law 328
Parseval's Theorem 492, 544
discrete form 498
Partial differential equations 818ff., 1332ff.
advective equation 826
alternating-direction implicit method (ADI) 847, 861f.
amplification factor 828, 834
analyze/factorize/operate package 824
artificial viscosity 831, 837
biconjugate gradient method 824
boundary conditions 819ff.
boundary value problems 819, 848
Cauchy problem 818f.
caution on high-order methods 844f.
Cayley's form 844
characteristics 818
Chebyshev acceleration 859f., 1332
classification of 818f.
comparison of rapid methods 854
conjugate gradient method 824
Courant condition 829, 832ff., 836
Courant condition (multidimensional) 846
Crank-Nicholson method 840, 842, 844, 846
cyclic reduction (CR) method 848f., 852ff.
diffusion equation 818, 838ff., 846, 855
Dirichlet boundary conditions 508, 820, 840, 850, 856, 858
elliptic, defined 818
error, varieties of 831ff.
explicit vs. implicit differencing 827
FACR method 854
finite difference method 821ff.
finite element methods 824
flux-conservative initial value problems 825ff.
forward Euler differencing 826f.
Forward Time Centered Space (FTCS) 827ff., 839ff., 843, 855
Fourier analysis and cyclic reduction (FACR) 848ff., 854
Gauss-Seidel method (relaxation) 855, 864ff., 876, 1338, 1341
Godunov's method 837
Helmholtz equation 852
hyperbolic 818, 825f.
implicit differencing 840
incomplete Cholesky conjugate gradient method (ICCG) 824
inhomogeneous boundary conditions 850f.
initial value problems 818f.
initial value problems, recommendations on 838ff.
Jacobi's method (relaxation) 855ff., 864
Laplace's equation 818
Lax method 828ff., 836, 845f.
Lax method (multidimensional) 845f.
matrix methods 824
mesh-drift instability 834f.
Monte Carlo methods 824
multidimensional initial value problems 844ff.
multigrid method 824, 862ff., 1009, 1334ff.
Neumann boundary conditions 508, 820, 840, 851, 858
nonlinear diffusion equation 842
nonlinear instability 831
numerical dissipation or viscosity 830
operator splitting 823, 847f., 861
outgoing wave boundary conditions 820
parabolic 818, 838ff.
parallel computing 1333
periodic boundary conditions 850, 858
piecewise parabolic method (PPM) 837
Poisson equation 818, 852
rapid (Fourier) methods 508ff., 824, 848ff.
relaxation methods 823, 854ff., 1332f.
Schrödinger equation 842ff.
second-order accuracy 833ff., 840
shock 831, 837
sparse matrices from 64
spectral methods 825
spectral radius 856ff., 862
stability vs. accuracy 830
stability vs. efficiency 821
staggered grids 513, 852
staggered leapfrog method 833f.
strongly implicit procedure 824

successive over-relaxation (SOR) 857ff., 862, 866, 1332f.
time splitting 847f., 861
two-step Lax-Wendroff method 835ff.
upwind differencing 832f., 837
variational methods 824
varieties of error 831ff.
von Neumann stability analysis 827f., 830, 833f., 840
wave equation 818, 825f.
see also Elliptic partial differential equations; Finite difference equations (FDEs)
Partial pivoting 29
Partition-exchange 323, 333
and pack() intrinsic function 1170
Partitioned matrix, inverse of 70
Party tricks 95ff., 168
Parzen window 547
Pascal, Numerical Recipes in 2/x, 2/xvii, 1
Pass-the-buck idiom 1102, 1128
Path integration, for function evaluation 201ff., 263, 1138
Pattern multiply of sparse matrices 74
PBCG (preconditioned biconjugate gradient method) 78f., 824
PC methods *see* Predictor-corrector methods
PCGPACK 71
PDEs *see* Partial differential equations
Pearson's r 630ff., 1276
PECE method 741
Pentagon, symmetries of 895
Percentile 320
Period of linear congruential generator 268
Periodic boundary conditions 850, 858
Periodogram 543ff., 566, 1258ff.
Lomb's normalized 569f., 574f., 1258ff.
variance of 544f.
Perl (programming language) 1/xvi
Perron's theorems, for convergence of recurrence relations 174f.
Perturbation methods for matrix inversion 65ff.
Phase error 831
Phase-locked loop 700
Phi statistic 625
Phillips-Twomey method 799ff.
Pi, computation of 906ff., 1352ff., 1357f.
Piecewise parabolic method (PPM) 837
Pincherle's theorem 175
Pivot element 29, 33, 757
in linear programming 428f.
Pivoting 27, 29ff., 46, 66, 90, 1014
full 29, 1014
implicit 30, 38, 1014, 1017
in LU decomposition 37f., 1017
partial 29, 33, 37f., 1017
and QR decomposition 92
in reduction to Hessenberg form 478
in relaxation method 757
as row and column operations 32
for tridiagonal systems 43
Pixel 519, 596, 803, 811
PL/1 2/x
Planck's constant 842
Plane rotation *see* Givens reduction; Jacobi transformation (or rotation)
Platykurtic distribution 606
Plotting of functions 342, 1182f.
POCS (projection onto convex sets) 805
Poetry 5f.
Pointer (Fortran 90) 2/xiiif., 938f., 944f., 953ff., 1197, 1212, 1266
as alias 939, 944f., 1286, 1333
allocating an array 941
allocating storage for derived type 955
for array of arrays 956, 1336
array of, forbidden 956, 1337
associated with target 938f., 944f., 952f., 1197
in Fortran 95 961
to function, forbidden 1067, 1210
initialization to null 2/xv, 961
returning array of unknown size 955f., 1184, 1259, 1261, 1327
undefined status 952f., 961, 1070, 1266, 1302
Poisson equation 519, 818, 852
Poisson probability function
cumulative 214
deviates from 281, 283ff., 571, 1154
semi-invariants of 608
tails compared to Gaussian 653
Poisson process 278, 282ff., 1153
Polak-Ribiere algorithm 390, 414ff., 1214
Poles *see* Complex plane, poles in
Polishing of roots 356, 363ff., 370f., 1193
poly() utility function 973, 977, 989, 998, 1072, 1096, 1192, 1258, 1284
Polymorphism 8
Polynomial interpolation 99, 102ff., 1043
Aitken's algorithm 102
in Bulirsch-Stoer method 724, 726, 1305
coefficients for 113ff., 1047f.
Lagrange's formula 84, 102f.
multidimensional 116ff., 1049ff.
Neville's algorithm 102f., 105, 134, 182, 1043
pathology in determining coefficients for 116
in predictor-corrector method 740
smoothing filters 645
see also Interpolation
Polynomials 167ff.
algebraic manipulations 169, 1072
approximate roots of Hermite polynomials 1062
approximate roots of Jacobi polynomials 1064
approximate roots of Laguerre polynomials 1061
approximating modified Bessel functions 230
approximation from Chebyshev coefficients 191, 1078f.
AUTODIN-II 890
CCITT 889f.
characteristic 368, 1193
characteristic, for digital filters 554, 559, 1257

characteristic, for eigenvalues of matrix 449, 469
Chebyshev 184ff., 1076ff.
coefficients from roots 980
CRC-16 890
cumulants of 977, 999, 1071f., 1192, 1365, 1378f.
deflation 362ff., 370f., 977
derivatives of 167, 978, 1071
division 84, 169, 362, 370, 977, 1072
evaluation of 167, 972, 977, 998f., 1071, 1258, 1365, 1376ff.
evaluation of derivatives 167, 978, 1071
extrapolation in Bulirsch-Stoer method 724, 726, 1305f.
extrapolation in Romberg integration 134
fitting 83, 114, 191, 645, 665, 674, 1078f., 1291
generator for CRC 889
ill-conditioned 362
masked evaluation of 1378
matrix method for roots 368, 1193
minimax 186, 198, 1076
monic 142f.
multiplication 169
operation count for 168
orthonormal 142, 184, 1009, 1061
parallel operations on 977ff., 998f., 1071f., 1192
primitive modulo 2 287ff., 301f., 889
roots of 178ff., 362ff., 368, 1191ff.
shifting of 192f., 978, 1079
stopping criterion in root finding 366
poly_term() utility function 974, 977, 989, 999, 1071f., 1192
Port, serial data 892
Portability 3, 963
Portable random number generator *see* Random number generator
Positive definite matrix, testing for 90
Positivity constraints 423
Postal Service (U.S.), barcode 894
PostScript 1/xvi, 1/xxiii, 2/xx
Powell's method 389, 402, 406ff., 1210ff.
Power (in a signal) 492f.
Power series 159ff., 167, 195
economization of 192f., 1061, 1080
Padé approximant of 194ff., 1080f.
Power spectral density *see* Fourier transform; Spectral density
Power spectrum estimation *see* Fourier transform; Spectral density
PowerStation, Microsoft Fortran 2/xix
PPM (piecewise parabolic method) 837
Precision
converting to double 1362
floating point 882, 937, 1343, 1361ff.
multiple 906ff., 1352ff., 1362
Preconditioned biconjugate gradient method (PBCG) 78f.
Preconditioning, in conjugate gradient methods 824
Predictor-corrector methods 702, 730, 740ff.
Adams-Bashforth-Moulton schemes 741
adaptive order methods 744
compared to other methods 740
fallacy of multiple correction 741
with fixed number of iterations 741
functional iteration vs. Newton's rule 742
multivalue compared with multistep 742ff.
starting and stopping 742, 744
stepsize control 742f.
present() intrinsic function 952
Prime numbers 915
Primitive polynomials modulo 2 287ff., 301f., 889
Principal directions 408f., 1210
Principal solution, of inverse problem 797
PRIVATE attribute 957, 1067
Prize, $1000 offered 272, 1141, 1150f.
Probability *see* Random number generator; Statistical tests
Probability density, change of variables in 278f.
Procedure *see* Program(s); Subprogram
Process loss 548
product() intrinsic function 948
Product Nystrom method 789, 1331
Program(s)
as black boxes 1/xviii, 6, 26, 52, 205, 341, 406
dependencies 921ff., 1434ff.
encapsulation 7
interfaces 2, 8
modularization 7f.
organization 5ff.
type declarations 2
typography of 2f., 12, 937
validation 3f.
Programming, serial vs. parallel 965, 987
Projection onto convex sets (POCS) 805
Projection operator, nonexpansive 805
Prolongation operator 864, 1337
Protocol, for communications 888
PSD (power spectral density) *see* Fourier transform; Spectral density
Pseudo-random numbers 266ff., 1141ff.
PUBLIC attribute 957, 1067
Puns, particularly bad 167, 744, 747
PURE attribute 2/xv, 960f., 964, 986
put_diag() utility function 985, 990, 1005, 1200
Pyramidal algorithm 586, 1264
Pythagoreans 392

QL *see* Eigensystems
QR *see* Eigensystems
QR decomposition 91f., 382, 386, 1039f., 1199
backsubstitution 92, 1040
and least squares 668
operation count 92
pivoting 92
updating 94, 382, 386, 1041, 1199
use for orthonormal basis 58, 94
Quadratic
convergence 49, 256, 351, 356, 409f., 419, 906
equations 20, 178, 391, 457

interpolation 353, 364
programming 436
Quadrature 123ff., 1052ff.
adaptive 123, 190, 788
alternative extended Simpson's rule 128
arbitrary weight function 151ff., 789, 1064, 1328
automatic 154
Bode's rule 126
change of variable in 137ff., 788, 1056ff.
by Chebyshev fitting 124, 189, 1078
classical formulas for 124ff.
Clenshaw-Curtis 124, 190, 512f.
closed formulas 125, 127f.
and computer science 881
by cubic splines 124
error estimate in solution 784
extended midpoint rule 129f., 135, 1054f.
extended rules 127ff., 134f., 786, 788ff., 1326, 1328
extended Simpson's rule 128
Fourier integrals 577ff., 1261ff.
Fourier integrals, infinite range 583
Gauss-Chebyshev 144, 512f.
Gauss-Hermite 144, 789, 1062
Gauss-Jacobi 144, 1063
Gauss-Kronrod 154
Gauss-Laguerre 144, 789, 1060
Gauss-Legendre 144, 783, 789, 1059, 1325
Gauss-Lobatto 154, 190, 512
Gauss-Radau 154
Gaussian integration 127, 140ff., 781, 783, 788f., 1009, 1059ff., 1325, 1328f.
Gaussian integration, nonclassical weight function 151ff., 788f., 1064f., 1328f.
for improper integrals 135ff., 789, 1055, 1328
for integral equations 781f., 786, 1325ff.
Monte Carlo 124, 155ff., 295ff., 306ff., 1161ff.
multidimensional 124, 155ff., 1052, 1065ff.
multidimensional, by recursion 1052, 1065
Newton-Cotes formulas 125ff., 140
open formulas 125ff., 129f., 135
related to differential equations 123
related to predictor-corrector methods 740
Romberg integration 124, 134f., 137, 182, 717, 788, 1054f., 1065, 1067
semi-open formulas 130
Simpson's rule 126, 133, 136f., 583, 782, 788ff., 1053
Simpson's three-eighths rule 126, 789f.
singularity removal 137ff., 788, 1057ff., 1328ff.
singularity removal, worked example 792, 1328ff.
trapezoidal rule 125, 127, 130ff., 134f., 579, 583, 782, 786, 1052ff., 1326f.
using FFTs 124
weight function $\log x$ 153
see also Integration of functions
Quadrature mirror filter 585, 593
Quantum mechanics, Uncertainty Principle 600
Quartile value 320
Quasi-Newton methods for minimization 390, 418ff., 1215
Quasi-random sequence 299ff., 318, 881, 888
Halton's 300
for Monte Carlo integration 304, 309, 318
Sobol's 300ff., 1160
see also Random number generator
Quicksort 320, 323ff., 330, 333, 1169f.
Quotient-difference algorithm 164

R-estimates 694
Radioactive decay 278
Radix base for floating point arithmetic 476, 882, 907, 913, 1231, 1343, 1357
Radix conversion 902, 906, 913, 1357
radix() intrinsic function 1231
Radix sort 1172
Ramanujan's identity for π 915
Random bits, generation of 287ff., 1159f.
Random deviates 266ff., 1141ff.
binomial 285f., 1155
exponential 278, 1151f.
gamma distribution 282f., 1153
Gaussian 267, 279f., 571, 798, 1152f.
normal 267, 279f., 571, 1152f.
Poisson 283ff., 571, 1154f.
quasi-random sequences 299ff., 881, 888, 1160f.
uniform 267ff., 1158f., 1166
uniform integer 270, 274ff.
Random number generator 266ff., 1141ff.
bitwise operations 287
Box-Muller algorithm 279, 1152
Data Encryption Standard 290ff., 1144, 1156ff.
good choices for modulus, multiplier and increment 274ff.
initializing 1144ff.
for integer-valued probability distribution 283f., 1154
integer vs. real implementation 273
L'Ecuyer's long period 271f.
lagged Fibonacci generator 1142, 1148ff.
linear congruential generator 267ff., 1142
machine language 269
Marsaglia shift register 1142, 1148ff.
Minimal Standard, Park and Miller's 269, 1142
nonrandomness of low-order bits 268f.
parallel 1009
perfect 272, 1141, 1150f.
planes, numbers lie on 268
portable 269ff., 1142
primitive polynomials modulo 2 287ff.
pseudo-DES 291, 1144, 1156ff.
quasi-random sequences 299ff., 881, 888, 1160f.
quick and dirty 274
quicker and dirtier 275
in Quicksort 324
random access to nth number 293

random bits 287ff., 1159f.
recommendations 276f.
rejection method 281ff.
serial 1141f.
shuffling procedure 270, 272
in simulated annealing method 438
spectral test 274
state space 1143f.
state space exhaustion 1141
subtractive method 273, 1143
system-supplied 267f.
timings 276f., 1151
transformation method 277ff.
trick for trigonometric functions 280
Random numbers *see* Monte Carlo; Random deviates
Random walk 20
random_number() intrinsic function 1141, 1143
random_seed() intrinsic function 1141
RANDU, infamous routine 268
Range 53f.
Rank (matrix) 53
kernel of finite 785
Rank (sorting) 320, 332, 1176
Rank (statistics) 633ff., 694f., 1277
Kendall's tau 637ff., 1279
Spearman correlation coefficient 634f., 1277ff.
sum squared differences of 634, 1277
Ratio variable (statistics) 623
Rational Chebyshev approximation 197ff., 1081f.
Rational function 99, 167ff., 194ff., 1080f.
approximation for Bessel functions 225
approximation for continued fraction 164, 211, 219f.
Chebyshev approximation 197ff., 1081f.
evaluation of 170, 1072f.
extrapolation in Bulirsch-Stoer method 718ff., 726, 1306f.
interpolation and extrapolation using 99, 104ff., 194ff., 718ff., 726
as power spectrum estimate 566
interpolation and extrapolation using 1043f., 1080ff., 1306
minimax 198
Re-entrant procedure 1052
real() intrinsic function, ambiguity of 947
Realizable (causal) 552, 554f.
reallocate() utility function 955, 990, 992, 1070, 1302
Rearranging *see* Sorting
Reciprocal, multiple precision 910f., 1355f.
Record, in data file 329
Recurrence relation 172ff., 971ff.
arithmetic progression 971f., 996
associated Legendre polynomials 247
Bessel function 172, 224, 227f., 234
binomial coefficients 209
Bulirsch-Stoer 105f.
characteristic polynomial of tridiagonal matrix 469
Clenshaw's recurrence formula 176f.
and continued fraction 175
continued fraction evaluation 164f.
convergence 175
cosine function 172, 500
cyclic reduction 974
dominant solution 174
exponential integrals 172
gamma function 206
generation of random bits 287f.
geometric progression 972, 996
Golden Mean 21
Legendre polynomials 172
minimal vs. dominant solution 174
modified Bessel function 232
Neville's 103, 182
orthonormal polynomials 142
Perron's theorems 174f.
Pincherle's theorem 175
for polynomial cumulants 977, 999, 1071f.
polynomial interpolation 103, 183
primitive polynomials modulo 2 287f.
random number generator 268
rational function interpolation 105f., 1043
recursive doubling 973, 977, 988, 999, 1071f., 1073
second order 974f., 1074
sequence of trig functions 173
sine function 172, 500
spherical harmonics 247
stability of 21, 173ff., 177, 224f., 227f., 232, 247, 975
trig functions 572
weight of Gaussian quadrature 144f.
Recursion
in Fortran 90 958
in multigrid method 865, 1009, 1336
Recursive doubling 973f., 979
cumulants of polynomial 977, 999, 1071f.
linear recurrences 973, 988, 1073
tridiagonal systems 976
RECURSIVE keyword 958, 1065, 1067
Recursive Monte Carlo integration 306ff., 1161
Recursive procedure 2/xiv, 958, 1065, 1067, 1166
as parallelization tool 958
base case 958
for multigrid method 1009, 1336
re-entrant 1052
Recursive stratified sampling 314ff., 1164ff.
Red-black *see* Odd-even ordering
Reduction functions 948ff.
Reduction of variance in Monte Carlo integration 299, 306ff.
References (explanation) 4f.
References (general bibliography) 916ff., 1359f.
Reflection formula for gamma function 206
Regula falsi (false position) 347ff., 1185f.
Regularity condition 775
Regularization
compared with optimal filtering 801
constrained linear inversion method 799ff.
of inverse problems 796ff.
linear 799ff.
nonlinear 813

objective criterion 802
Phillips-Twomey method 799ff.
Tikhonov-Miller 799ff.
trade-off curve 799
two-dimensional 803
zeroth order 797
see also Inverse problems
Regularizing operator 798
Reid, John 2/xiv, 2/xvi
Rejection method for random number generator 281ff.
Relaxation method
for algebraically difficult sets 763
automated allocation of mesh points 774f., 777
computation of spheroidal harmonics 764ff., 1319ff.
for differential equations 746f., 753ff., 1316ff.
elliptic partial differential equations 823, 854ff., 1332f.
example 764ff., 1319ff.
Gauss-Seidel method 855, 864ff., 876, 1338, 1341
internal boundary conditions 775ff.
internal singular points 775ff.
Jacobi's method 855f., 864
successive over-relaxation (SOR) 857ff., 862, 866, 1332f.
see also Multigrid method
Remes algorithms
exchange algorithm 553
for minimax rational function 199
reshape() intrinsic function 950
communication bottleneck 969
order keyword 1050, 1246
Residual 49, 54, 78
in multigrid method 863, 1338
Resolution function, in Backus-Gilbert method 807
Response function 531
Restriction operator 864, 1337
RESULT keyword 958, 1073
Reward, $1000 offered 272, 1141, 1150f.
Richardson's deferred approach to the limit 134, 137, 182, 702, 718ff., 726, 788, 869
see also Bulirsch-Stoer method
Richtmyer artificial viscosity 837
Ridders' method, for numerical derivatives 182, 1075
Ridders' method, root finding 341, 349, 351, 1187
Riemann shock problem 837
Right eigenvalues and eigenvectors 451
Rise/fall time 548f.
Robust estimation 653, 694ff., 700, 1294
Andrew's sine 697
average deviation 605
double exponential errors 696
Kalman filtering 700
Lorentzian errors 696f.
mean absolute deviation 605
nonparametric correlation 633ff., 1277
Tukey's biweight 697
use of a priori covariances 700
see also Statistical tests
Romberg integration 124, 134f., 137, 182, 717, 788, 1054f., 1065
Root finding 143, 340ff., 1009, 1059
advanced implementations of Newton's rule 386
Bairstow's method 364, 370, 1193
bisection 343, 346f., 352f., 359, 390, 469, 698, 1184f.
bracketing of roots 341, 343ff., 353f., 362, 364, 369, 1183f.
Brent's method 341, 349, 660f., 1188f., 1286
Broyden's method 373, 382f., 386, 1199
compared with multidimensional minimization 375
complex analytic functions 364
in complex plane 204
convergence criteria 347, 374
deflation of polynomials 362ff., 370f., 1192
without derivatives 354
double root 341
eigenvalue methods 368, 1193
false position 347ff., 1185f.
Jenkins-Traub method 369
Laguerre's method 341, 366f., 1191f.
Lehmer-Schur algorithm 369
Maehly's procedure 364, 371
matrix method 368, 1193
Muller's method 364, 372
multiple roots 341
Newton's rule 143f., 180, 341, 355ff., 362, 364, 370, 372ff., 376, 469, 740, 749f., 754, 787, 874, 876, 911f., 1059, 1189, 1194, 1196, 1314ff., 1339, 1341, 1355f.
pathological cases 343, 356, 362, 372
polynomials 341, 362ff., 449, 1191f.
in relaxation method 754, 1316
Ridders' method 341, 349, 351, 1187
root-polishing 356, 363ff., 369ff., 1193
safe Newton's rule 359, 1190
secant method 347ff., 358, 364, 399, 1186f.
in shooting method 746, 749f., 1314f.
singular Jacobian in Newton's rule 386
stopping criterion for polynomials 366
use of minimum finding 341
using derivatives 355ff., 1189
zero suppression 372
see also Roots
Root polishing 356, 363ff., 369ff., 1193
Roots
Chebyshev polynomials 184
complex *n*th root of unity 999f., 1379
cubic equations 179f.
Hermite polynomials, approximate 1062
Jacobi polynomials, approximate 1064
Laguerre polynomials, approximate 1061
multiple 341, 364ff., 1192
nonlinear equations 340ff.
polynomials 341, 362ff., 449, 1191f.
quadratic equations 178

reflection in unit circle 560, 1257
square, multiple precision 912, 1356
see also Root finding
Rosenbrock method 730, 1308
compared with semi-implicit extrapolation 739
stepsize control 731, 1308f.
Roundoff error 20, 881, 1362
bracketing a minimum 399
compile time vs. run time 1012
conjugate gradient method 824
eigensystems 458, 467, 470, 473, 476, 479, 483
extended trapezoidal rule 132
general linear least squares 668, 672
graceful 883, 1343
hardware aspects 882, 1343
Householder reduction 466
IEEE standard 882f., 1343
interpolation 100
least squares fitting 658, 668
Levenberg-Marquardt method 679
linear algebraic equations 23, 27, 29, 47, 56, 84, 1022
linear predictive coding (LPC) 564
magnification of 20, 47, 1022
maximum entropy method (MEM) 567
measuring 881f., 1343
multidimensional minimization 418, 422
multiple roots 362
numerical derivatives 180f.
recurrence relations 173
reduction to Hessenberg form 479
series 164f.
straight line fitting 658
variance 607
Row degeneracy 22
Row-indexed sparse storage 71f., 1030
transpose 73f.
Row operations on matrix 28, 31f.
Row totals 624
RSS algorithm 314ff., 1164
RST properties (reflexive, symmetric, transitive) 338
Runge-Kutta method 702, 704ff., 731, 740, 1297ff., 1308
Cash-Karp parameters 710, 1299f.
embedded 709f., 731, 1298, 1308
high-order 705
quality control 722
stepsize control 708ff.
Run-length encoding 901
Runge-Kutta method
high-order 1297
stepsize control 1298f.
Rybicki, G.B. 84ff., 114, 145, 252, 522, 574, 600

S-box for Data Encryption Standard 1148
Sampling
importance 306f.
Latin square or hypercube 305f.
recursive stratified 314ff., 1164
stratified 308f.
uneven or irregular 569, 648f., 1258
Sampling theorem 495, 543
for numerical approximation 600ff.
Sande-Tukey FFT algorithm 503
SAVE attribute 953f., 958f., 961, 1052, 1070, 1266, 1293
redundant use of 958f.
SAVE statements 3
Savitzky-Golay filters
for data smoothing 644ff., 1283f.
for numerical derivatives 183, 645
scale() intrinsic function 1107
Scallop loss 548
Scatter-with-combine functions 984, 1002f., 1032, 1366, 1380f.
scatter_add() utility function 984, 990, 1002, 1032
scatter_max() utility function 984, 990, 1003
Schonfelder, Lawrie 2/xi
Schrage's algorithm 269
Schrödinger equation 842ff.
Schultz's method for matrix inverse 49, 598
Scope 956ff., 1209, 1293, 1296
Scoping unit 939
SDLC checksum 890
Searching
with correlated values 111, 1046f.
an ordered table 110f., 1045f.
selection 333, 1177f.
Secant method 341, 347ff., 358, 364, 399, 1186f.
Broyden's method 382f., 1199f.
multidimensional (Broyden's) 373, 382f., 1199
Second Euler-Maclaurin summation formula 135f.
Second order differential equations 726, 1307
Seed of random number generator 267, 1146f.
select case statement 2/xiv, 1010, 1036
Selection 320, 333, 1177f.
find m largest elements 336, 1179f.
heap algorithm 336, 1179
for median 698, 1294
operation count 333
by packing 1178
parallel algorithms 1178
by partition-exchange 333, 1177f.
without rearrangement 335, 1178f.
timings 336
use to find median 609
Semi-implicit Euler method 730, 735f.
Semi-implicit extrapolation method 730, 735f., 1310f.
compared with Rosenbrock method 739
stepsize control 737, 1311f.
Semi-implicit midpoint rule 735f., 1310f.
Semi-invariants of a distribution 608
Sentinel, in Quicksort 324, 333
Separable kernel 785
Separation of variables 246
Serial computing
convergence of quadrature 1060
random numbers 1141
sorting 1167
Serial data port 892

Series 159ff.
accelerating convergence of 159ff.
alternating 160f., 1070
asymptotic 161
Bessel function K_ν 241
Bessel function Y_ν 235
Bessel functions 160, 223
cosine integral 250
divergent 161
economization 192f., 195, 1080
Euler's transformation 160f., 1070
exponential integral 216, 218
Fresnel integral 248
hypergeometric 202, 263, 1138
incomplete beta function 219
incomplete gamma function 210, 1090f.
Laurent 566
relation to continued fractions 163f.
roundoff error in 164f.
sine and cosine integrals 250
sine function 160
Taylor 355f., 408, 702, 709, 754, 759
transformation of 160ff., 1070
van Wijngaarden's algorithm 161, 1070
Shaft encoder 886
Shakespeare 9
Shampine's Rosenbrock parameters 732, 1308
shape() intrinsic function 938, 949
Shell algorithm (Shell's sort) 321ff., 1168
Sherman-Morrison formula 65ff., 83, 382
Shifting of eigenvalues 449, 470f., 480
Shock wave 831, 837
Shooting method
computation of spheroidal harmonics 772, 1321ff.
for differential equations 746, 749ff., 770ff., 1314ff., 1321ff.
for difficult cases 753, 1315f.
example 770ff., 1321ff.
interior fitting point 752, 1315f., 1323ff.
Shuffling to improve random number generator 270, 272
Side effects
prevented by data hiding 957, 1209, 1293, 1296
and PURE subprograms 960
Sidelobe fall-off 548
Sidelobe level 548
sign() intrinsic function, modified in Fortran 95 961
Signal, bandwidth limited 495
Significance (numerical) 19
Significance (statistical) 609f.
one- vs. two-sided 632
peak in Lomb periodogram 570
of 2-d K-S test 640, 1281
two-tailed 613
SIMD machines (Single Instruction Multiple Data) 964, 985f., 1009, 1084f.
Similarity transform 452ff., 456, 476, 478, 482
Simplex
defined 402
method in linear programming 389, 402, 423ff., 431ff., 1216ff.
method of Nelder and Mead 389, 402ff., 444, 697f., 1208f., 1222ff.
use in simulated annealing 444, 1222ff.
Simpson's rule 124ff., 128, 133, 136f., 583, 782, 788f., 1053f.
Simpson's three-eighths rule 126, 789f.
Simulated annealing *see* Annealing, method of simulated
Simulation *see* Monte Carlo
Sine function
evaluated from $\tan(\theta/2)$ 173
recurrence 172
series 160
Sine integral 248, 250ff., 1123, 1125f.
continued fraction 250
series 250
see also Cosine integral
Sine transform *see* Fast Fourier transform (FFT); Fourier transform
Singleton's algorithm for FFT 525
Singular value decomposition (SVD) 23, 25, 51ff., 1022
approximation of matrices 58f.
backsubstitution 56, 1022f.
and bases for nullspace and range 53
confidence levels from 693f.
covariance matrix 693f.
fewer equations than unknowns 57
for inverse problems 797
and least squares 54ff., 199f., 668, 670ff., 1081, 1290f.
in minimization 410
more equations than unknowns 57f.
parallel algorithms 1026
and rational Chebyshev approximation 199f., 1081f.
of square matrix 53ff., 1023
use for ill-conditioned matrices 56, 58, 449
use for orthonormal basis 58, 94
Singularities
of hypergeometric function 203, 263
in integral equations 788ff., 1328
in integral equations, worked example 792, 1328ff.
in integrands 135ff., 788, 1055, 1328ff.
removal in numerical integration 137ff., 788, 1057ff., 1328ff.
Singularity, subtraction of the 789
SIPSOL 824
Six-step framework, for FFT 983, 1240
size() intrinsic function 938, 942, 945, 948
Skew array section 2/xii, 945, 960, 985, 1284
Skewness of distribution 606, 608, 1269
Smoothing
of data 114, 644ff., 1283f.
of data in integral equations 781
importance in multigrid method 865
sn function 261, 1137f.
Snyder, N.L. 1/xvi
Sobol's quasi-random sequence 300ff., 1160f.
Sonata 9
Sonnet 9
Sorting 320ff., 1167ff.
bubble sort 1168

bubble sort cautioned against 321
compared to selection 333
covariance matrix 669, 681, 1289
eigenvectors 461f., 1227
Heapsort 320, 327f., 336, 1171f., 1179
index table 320, 329f., 1170, 1173ff., 1176
operation count 320ff.
by packing 1171
parallel algorithms 1168, 1171f., 1176
Quicksort 320, 323ff., 330, 333, 1169f.
radix sort 1172
rank table 320, 332, 1176
ranking 329, 1176
by reshaping array slices 1168
Shell's method 321ff., 1168
straight insertion 321f., 461f., 1167, 1227
SP, defined 937
SPARC or SPARCstation 1/xxii, 2/xix, 4
Sparse linear equations 23, 63ff., 732, 1030
band diagonal 43, 1019ff.
biconjugate gradient method 77, 599, 1034
data type for 1030
indexed storage 71f., 1030
in inverse problems 804
minimum residual method 78
named patterns 64, 822
partial differential equations 822ff.
relaxation method for boundary value problems 754, 1316
row-indexed storage 71f., 1030
wavelet transform 584, 598
see also Matrix
Spearman rank-order coefficient 634f., 694f., 1277
Special functions *see* Function
Spectral analysis *see* Fourier transform; Periodogram
Spectral density 541
and data windowing 545ff.
figures of merit for data windows 548f.
normalization conventions 542f.
one-sided PSD 492
periodogram 543ff., 566, 1258ff.
power spectral density (PSD) 492f.
power spectral density per unit time 493
power spectrum estimation by FFT 542ff., 1254ff.
power spectrum estimation by MEM 565ff., 1258
two-sided PSD 493
variance reduction in spectral estimation 545
Spectral lines, how to smooth 644
Spectral methods for partial differential equations 825
Spectral radius 856ff., 862
Spectral test for random number generator 274
Spectrum *see* Fourier transform
Spherical Bessel functions 234
routine for 245, 1121
Spherical harmonics 246ff.
orthogonality 246
routine for 247f., 1122
stable recurrence for 247
table of 246
see also Associated Legendre polynomials
Spheroidal harmonics 764ff., 770ff., 1319ff.
boundary conditions 765
normalization 765
routine for 768ff., 1319ff., 1323ff.
Spline 100
cubic 107ff., 1044f.
gives tridiagonal system 109
natural 109, 1044f.
operation count 109
two-dimensional (bicubic) 120f., 1050f.
spread() intrinsic function 945, 950, 969, 1000, 1094, 1290f.
and dimensional expansion 966ff.
Spread matrix 808
Spread spectrum 290
Square root, complex 172
Square root, multiple precision 912, 1356f.
Square window 546, 1254ff.
SSP (small-scale parallel) machines 965ff., 972, 974, 984, 1011, 1016ff., 1021, 1059f., 1226ff., 1250
Stability 20f.
of Clenshaw's recurrence 177
Courant condition 829, 832ff., 836, 846
diffusion equation 840
of Gauss-Jordan elimination 27, 29
of implicit differencing 729, 840
mesh-drift in PDEs 834f.
nonlinear 831, 837
partial differential equations 820, 827f.
of polynomial deflation 363
in quadrature solution of Volterra equation 787f.
of recurrence relations 173ff., 177, 224f., 227f., 232, 247
and stiff differential equations 728f.
von Neumann analysis for PDEs 827f., 830, 833f., 840
see also Accuracy
Stabilized Kolmogorov-Smirnov test 621
Stabilizing functional 798
Staggered leapfrog method 833f.
Standard (probable) errors 1288, 1290
Standard deviation
of a distribution 605, 1269
of Fisher's z 632
of linear correlation coefficient 630
of sum squared difference of ranks 635, 1277
Standard (probable) errors 610, 656, 661, 667, 671, 684
Stars, as text separator 1009
Statement function, superseded by internal subprogram 1057, 1256
Statement labels 9
Statistical error 653
Statistical tests 603ff., 1269ff.
Anderson-Darling 621
average deviation 605, 1269
bootstrap method 686f.
chi-square 614f., 623ff., 1272, 1275f.

contingency coefficient C 625, 1275
contingency tables 622ff., 638, 1275f.
correlation 603f.
Cramer's V 625, 1275
difference of distributions 614ff., 1272
difference of means 609ff., 1269f.
difference of variances 611, 613, 1271
entropy measures of association 626ff., 1275f.
F-test 611, 613, 1271
Fisher's z-transformation 631f., 1276
general paradigm 603
Kendall's tau 634, 637ff., 1279
Kolmogorov-Smirnov 614, 617ff., 640, 694, 1273f., 1281
Kuiper's statistic 621
kurtosis 606, 608, 1269
L-estimates 694
linear correlation coefficient 630ff., 1276
M-estimates 694ff.
mean 603ff., 608ff., 1269f.
measures of association 604, 622ff., 1275
measures of central tendency 604ff., 1269
median 605, 694
mode 605
moments 604ff., 608, 1269
nonparametric correlation 633ff., 1277
Pearson's r 630ff., 1276
for periodic signal 570
phi statistic 625
R-estimates 694
rank correlation 633ff., 1277
robust 605, 634, 694ff.
semi-invariants 608
for shift vs. for spread 620f.
significance 609f., 1269ff.
significance, one- vs. two-sided 613, 632
skewness 606, 608, 1269
Spearman rank-order coefficient 634f., 694f., 1277
standard deviation 605, 1269
strength vs. significance 609f., 622
Student's t 610, 631, 1269
Student's t, for correlation 631
Student's t, paired samples 612, 1271
Student's t, Spearman rank-order coefficient 634, 1277
Student's t, unequal variances 611, 1270
sum squared difference of ranks 635, 1277
Tukey's trimean 694
two-dimensional 640, 1281ff.
variance 603ff., 607f., 612f., 1269ff.
Wilcoxon 694
see also Error; Robust estimation
Steak, without sizzle 809
Steed's method
Bessel functions 234, 239
continued fractions 164f.
Steepest descent method 414
in inverse problems 804
Step
doubling 130, 708f., 1052
tripling 136, 1055
Stieltjes, procedure of 151
Stiff equations 703, 727ff., 1308ff.
Kaps-Rentrop method 730, 1308
methods compared 739
predictor-corrector method 730
r.h.s. independent of x 729f.
Rosenbrock method 730, 1308
scaling of variables 730
semi-implicit extrapolation method 730, 1310f.
semi-implicit midpoint rule 735f., 1310f.
Stiff functions 100, 399
Stirling's approximation 206, 812
Stoermer's rule 726, 1307
Stopping criterion, in multigrid method 875f.
Stopping criterion, in polynomial root finding 366
Storage
band diagonal matrix 44, 1019
sparse matrices 71f., 1030
Storage association 2/xiv
Straight injection 867
Straight insertion 321f., 461f., 1167, 1227
Straight line fitting 655ff., 667f., 1285ff.
errors in both coordinates 660ff., 1286ff.
robust estimation 698, 1294ff.
Strassen's fast matrix algorithms 96f.
Stratified sampling, Monte Carlo 308f., 314
Stride (of an array) 944
communication bottleneck 969
Strongly implicit procedure (SIPSOL) 824
Structure constructor 2/xii
Structured programming 5ff.
Student's probability distribution 221f.
Student's t-test
for correlation 631
for difference of means 610, 1269
for difference of means (paired samples) 612, 1271
for difference of means (unequal variances) 611, 1270
for difference of ranks 635, 1277
Spearman rank-order coefficient 634, 1277
Sturmian sequence 469
Sub-random sequences *see* Quasi-random sequence
Subprogram 938
for data hiding 957, 1209, 1293, 1296
internal 954, 957, 1057, 1067, 1226, 1256
in module 940
undefined variables on exit 952f., 961, 1070, 1266, 1293, 1302
Subscript triplet (for array) 944
Subtraction, multiple precision 907, 1353
Subtractive method for random number generator 273, 1143
Subvector scaling 972, 974, 996, 1000
Successive over-relaxation (SOR) 857ff., 862, 1332f.
bad in multigrid method 866
Chebyshev acceleration 859f., 1332f.
choice of overrelaxation parameter 858
with logical mask 1333f.
parallelization 1333
sum() intrinsic function 945, 948, 966

Sum squared difference of ranks 634, 1277
Sums *see* Series
Sun 1/xxii, 2/xix, 886
SPARCstation 1/xxii, 2/xix, 4
Supernova 1987A 640
SVD *see* Singular value decomposition (SVD)
swap() utility function 987, 990f., 1015, 1210
Symbol, of operator 866f.
Synthetic division 84, 167, 362, 370
parallel algorithms 977ff., 999, 1048, 1071f., 1079, 1192
repeated 978f.
Systematic errors 653

Tableau (interpolation) 103, 183
Tangent function, continued fraction 163
Target, for pointer 938f., 945, 952f.
Taylor series 180, 355f., 408, 702, 709, 742, 754, 759
Test programs 3
Thermodynamics, analogy for simulated annealing 437
Thinking Machines, Inc. 964
Threshold multiply of sparse matrices 74, 1031
Tides 560f.
Tikhonov-Miller regularization 799ff.
Time domain 490
Time splitting 847f., 861
tiny() intrinsic function 952
Toeplitz matrix 82, 85ff., 195, 1038
LU decomposition 87
new, fast algorithms 88f.
nonsymmetric 86ff., 1038
Tongue twisters 333
Torus 297f., 304
Trade-off curve 795, 809
Trademarks 1/xxii, 2/xixf.
Transformation
Gauss 256
Landen 256
method for random number generator 277ff.
Transformational functions 948ff.
Transforms, number theoretic 503f.
Transport error 831ff.
transpose() intrinsic function 950, 960, 969, 981, 1050, 1246
Transpose of sparse matrix 73f.
Trapezoidal rule 125, 127, 130ff., 134f., 579, 583, 782, 786, 1052, 1326f.
Traveling salesman problem 438ff., 1219ff.
Tridiagonal matrix 42, 63, 150, 453f., 488, 839f., 1018f.
in alternating-direction implicit method (ADI) 861f.
from cubic spline 109
cyclic 67, 1030
in cyclic reduction 853
eigenvalues 469ff., 1228
with fringes 822
from operator splitting 861f.
parallel algorithm 975, 1018, 1229f.
recursive splitting 1229f.
reduction of symmetric matrix to 462ff., 470, 1227f.
serial algorithm 1018f.
see also Matrix
Trigonometric
functions, linear sequences 173
functions, recurrence relation 172, 572
functions, $\tan(\theta/2)$ as minimal 173
interpolation 99
solution of cubic equation 179f.
Truncation error 20f., 399, 709, 881, 1362
in multigrid method 875
in numerical derivatives 180
Tukey's biweight 697
Tukey's trimean 694
Turbo Pascal (Borland) 8
Twin errors 895
Two-dimensional *see* Multidimensional
Two-dimensional K–S test 640, 1281ff.
Two-pass algorithm for variance 607, 1269
Two-point boundary value problems 702, 745ff., 1314ff.
automated allocation of mesh points 774f., 777
boundary conditions 745ff., 749, 751f., 771, 1314ff.
difficult cases 753, 1315f.
eigenvalue problem for differential equations 748, 764ff., 770ff., 1319ff.
free boundary problem 748, 776
grid (mesh) points 746f., 754, 774f., 777
internal boundary conditions 775ff.
internal singular points 775ff.
linear requires no iteration 751
multiple shooting 753
problems reducible to standard form 748
regularity condition 775
relaxation method 746f., 753ff., 1316ff.
relaxation method, example of 764ff., 1319
shooting to a fitting point 751ff., 1315f., 1323ff.
shooting method 746, 749ff., 770ff., 1314ff., 1321ff.
shooting method, example of 770ff., 1321ff.
singular endpoints 751, 764, 771, 1315f., 1319ff.
see also Elliptic partial differential equations
Two-sided exponential error distribution 696
Two-sided power spectral density 493
Two-step Lax-Wendroff method 835ff.
Two-volume edition, plan of 1/xiii
Two's complement arithmetic 1144
Type declarations, explicit vs. implicit 2

Ubound() intrinsic function 949
ULTRIX 1/xxiii, 2/xix
Uncertainty coefficient 628
Uncertainty principle 600
Undefined status, of arrays and pointers 952f., 961, 1070, 1266, 1293, 1302
Underflow, in IEEE arithmetic 883, 1343
Underrelaxation 857
Uniform deviates *see* Random deviates, uniform

Unitary (function) 843f.
Unitary (matrix) *see* Matrix
unit_matrix() utility function 985, 990, 1006, 1216, 1226, 1325
UNIX 1/xxiii, 2/viii, 2/xix, 4, 17, 276, 293, 886
Upper Hessenberg matrix *see* Hessenberg matrix
U.S. Postal Service barcode 894
unpack() intrinsic function 950, 964
  communication bottleneck 969
Upper subscript 944
upper_triangle() utility function 990, 1006, 1226, 1305
Upwind differencing 832f., 837
USE statement 936, 939f., 954, 957, 1067, 1384
USES keyword in program listings 2
Utility functions 987ff., 1364ff.
  add vector to matrix diagonal 1004, 1234, 1366, 1381
  alphabetical listing 988ff.
  argument checking 994f., 1370f.
  arithmetic progression 996, 1072, 1127, 1365, 1371f.
  array reallocation 992, 1070f., 1365, 1368f.
  assertion of numerical equality 995, 1022, 1365, 1370f.
  compared to intrinsics 990ff.
  complex *n*th root of unity 999f., 1379
  copying arrays 991, 1034, 1327f., 1365f.
  create unit matrix 1006, 1382
  cumulative product of an array 997f., 1072, 1086, 1375
  cumulative sum of an array 997, 1280f., 1365, 1375
  data types 1361
  elemental functions 1364
  error handling 994f., 1036, 1370f.
  generic functions 1364
  geometric progression 996f., 1365, 1372ff.
  get diagonal of matrix 1005, 1226f., 1366, 1381f.
  length of a vector 1008, 1383
  linear recurrence 996
  location in an array 992ff., 1015, 1017ff.
  location of first logical "true" 993, 1041, 1369
  location of maximum array value 993, 1015, 1017, 1365, 1369
  location of minimum array value 993, 1369f.
  logical assertion 994, 1086, 1090, 1092, 1365, 1370
  lower triangular mask 1007, 1200, 1382
  masked polynomial evaluation 1378
  masked swap of elements in two arrays 1368
  moving data 990ff., 1015
  multiply vector into matrix diagonal 1004f., 1366, 1381
  nrutil.f90 (module file) 1364ff.
  outer difference of vectors 1001, 1366, 1380
  outer logical and of vectors 1002
  outer operations on vectors 1000ff., 1379f.
  outer product of vectors 1000f., 1076, 1365f., 1379
  outer quotient of vectors 1001, 1379
  outer sum of vectors 1001, 1379f.
  overloading 1364
  partial cumulants of a polynomial 999, 1071, 1192f., 1365, 1378f.
  polynomial evaluation 996, 998f., 1258, 1365, 1376ff.
  scatter-with-add 1002f., 1032f., 1366, 1380f.
  scatter-with-combine 1002f., 1032f., 1380f.
  scatter-with-max 1003f., 1366, 1381
  set diagonal elements of matrix 1005, 1200, 1366, 1382
  skew operation on matrices 1004ff., 1381ff.
  swap elements of two arrays 991, 1015, 1365ff.
  upper triangular mask 1006, 1226, 1305, 1382

V-cycle 865, 1336
vabs() utility function 990, 1008, 1290
Validation of Numerical Recipes procedures 3f.
Valley, long or narrow 403, 407, 410
Van Cittert's method 804
Van Wijngaarden-Dekker-Brent method *see* Brent's method
Vandermonde matrix 82ff., 114, 1037, 1047
Variable length code 896, 1346ff.
Variable metric method 390, 418ff., 1215
  compared to conjugate gradient method 418
Variable step-size integration 123, 135, 703, 707ff., 720, 726, 731, 737, 742ff., 1298ff., 1303, 1308f., 1311ff.
Variance(s)
  correlation 605
  of distribution 603ff., 608, 611, 613, 1269
  pooled 610
  reduction of (in Monte Carlo) 299, 306ff.
  statistical differences between two 609, 1271
  two-pass algorithm for computing 607, 1269
  see also Covariance
Variational methods, partial differential equations 824
VAX 275, 293
Vector(s)
  length 1008, 1383
  norms 1036
  outer difference 1001, 1366, 1380
  outer operations 1000ff., 1379f.
  outer product 1000f., 1076, 1365f., 1379
Vector reduction 972, 977, 998
Vector subscripts 2/xiif., 984, 1002, 1032, 1034
  communication bottleneck 969, 981, 1250
VEGAS algorithm for Monte Carlo 309ff., 1161
Verhoeff's algorithm for checksums 894f., 1345

Viète's formulas for cubic roots 179
Vienna Fortran 2/xv
Virus, computer 889
Viscosity
  artificial 831, 837
  numerical 830f., 837
Visibility 956ff., 1209, 1293, 1296
VMS 1/xxii, 2/xix
Volterra equations 780f., 1326
  adaptive stepsize control 788
  analogy with ODEs 786
  block-by-block method 788
  first kind 781, 786
  nonlinear 781, 787
  second kind 781, 786ff., 1326f.
  unstable quadrature 787f.
von Neuman, John 963, 965
von Neumann-Richtmyer artificial viscosity 837
von Neumann stability analysis for PDEs 827f., 830, 833f., 840
Vowellish (coding example) 896f., 902

W-cycle 865, 1336
Warranty, disclaimer of 1/xx, 2/xvii
Wave equation 246, 818, 825f.
Wavelet transform 584ff., 1264ff.
  appearance of wavelets 590ff.
  approximation condition of order p 585
  coefficient values 586, 589, 1265
  contrasted with Fourier transform 584, 594
  Daubechies wavelet filter coefficients 584ff., 588, 590f., 594, 598, 1264ff.
  detail information 585
  discrete wavelet transform (DWT) 586f., 1264
  DWT (discrete wavelet transform) 586f., 1264ff.
  eliminating wrap-around 587
  fast solution of linear equations 597ff.
  filters 592f.
  and Fourier domain 592f.
  image processing 596f.
  for integral equations 782
  inverse 587
  Lemarie's wavelet 593
  of linear operator 597ff.
  mother-function coefficient 587
  mother functions 584
  multidimensional 595, 1267f.
  nonsmoothness of wavelets 591
  pyramidal algorithm 586, 1264
  quadrature mirror filter 585
  smooth information 585
  truncation 594f.
  wavelet filter coefficient 584, 587
  wavelets 584, 590ff.
Wavelets *see* Wavelet transform
Weber function 204
Weighted Kolmogorov-Smirnov test 621
Weighted least-squares fitting *see* Least squares fitting
Weighting, full vs. half in multigrid 867
Weights for Gaussian quadrature 140ff., 788f., 1059ff., 1328f.
  nonclassical weight function 151ff., 788f., 1064f., 1328f.
Welch window 547, 1254ff.
WG5 (ISO/IEC JTC1/SC22/WG5 Committee) 2/xiff.
where construct 943, 1291
  contrasted with merge 1023
  for iteration of a vector 1060
  nested 2/xv, 943, 960, 1100
  not MIMD 985
While iteration 13
Wiener filtering 535, 539ff., 558, 644
  compared to regularization 801
Wiener-Khinchin theorem 492, 558, 566f.
Wilcoxon test 694
Window function
  Bartlett 547, 1254ff.
  flat-topped 549
  Hamming 547
  Hann 547
  Parzen 547
  square 544, 546, 1254ff.
  Welch 547, 1254ff.
Windowing for spectral estimation 1255f.
Windows 95 2/xix
Windows NT 2/xix
Winograd Fourier transform algorithms 503
Woodbury formula 68ff., 83
Wordlength 18
Workspace, reallocation in Fortran 90 1070f.
World Wide Web, Numerical Recipes site 1/xx, 2/xvii
Wraparound
  in integer arithmetic 1146, 1148
  order for storing spectrum 501
  problem in convolution 533
Wronskian, of Bessel functions 234, 239

X.25 protocol 890
X3J3 Committee 2/viii, 2/xff., 2/xv, 947, 959, 964, 968, 990
XMODEM checksum 889
X-ray diffraction pattern, processing of 805

Yale Sparse Matrix Package 64, 71

Z-transform 554, 559, 565
Z-transformation, Fisher's 631f., 1276
Zaman, A. 1149
Zealots 814
Zebra relaxation 866
Zero contours 372
Zero-length array 944
Zeroth-order regularization 796ff.
Zip code, barcode for 894
Ziv-Lempel compression 896
zroots_unity() utility function 974, 990, 999

Fortran 90 Code Chapters B1–B20

Fortran 90 versions of all the Numerical Recipes routines appear in the following Chapters B1 through B20, numbered in correspondence with Chapters 1 through 20 in Volume 1. Within each chapter, the routines appear in the same order as in Volume 1, but not broken out separately by section number within Volume 1's chapters.

There are commentaries accompanying many of the routines, generally following the printed listing of the routine to which they apply. These are of two kinds: issues related to parallelizing the algorithm in question, and issues related to the Fortran 90 implementation. To distinguish between these two, rather different, kinds of discussions, we use the two icons,

the left icon (above) indicating a "parallel note," and the right icon denoting a "Fortran 90 tip." Specific code segments of the routine that are discussed in these commentaries are singled out by reproducing some of the code as an "index line" next to the icon, or at the beginning of subsequent paragraphs if there are several items that are commented on.

`d=merge(FPMIN,d,abs(d)<FPMIN)` This would be the start of a discussion of code that begins at the line in the listing containing the indicated code fragment.

⋆ ⋆ ⋆

A row of stars, like the above, is used between unrelated routines, or at the beginning and end of related groups of routines.

Some chapters contain discussions that are more general than commentary on individual routines, but that were deemed too specific for inclusion in Chapters 21 through 23. Here are some highlights of this additional material:

- Approximations to roots of orthogonal polynomials for parallel computation of Gaussian quadrature formulas (Chapter B4)
- Difficulty of, and tricks for, parallel calculation of special function values in a SIMD model of computation (Chapter B6)
- Parallel random number generation (Chapter B7)
- Fortran 90 tricks for dealing with ties in sorted arrays, counting things in boxes, etc. (Chapter B14)
- Use of recursion in implementing multigrid elliptic PDE solvers (Chapter B19)

Chapter B1. Preliminaries

```
SUBROUTINE flmoon(n,nph,jd,frac)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,nph
INTEGER(I4B), INTENT(OUT) :: jd
REAL(SP), INTENT(OUT) :: frac
```

Our programs begin with an introductory comment summarizing their purpose and explaining their calling sequence. This routine calculates the phases of the moon. Given an integer `n` and a code `nph` for the phase desired (`nph` = 0 for new moon, 1 for first quarter, 2 for full, 3 for last quarter), the routine returns the Julian Day Number `jd`, and the fractional part of a day `frac` to be added to it, of the `n`th such phase since January, 1900. Greenwich Mean Time is assumed.

```
REAL(SP), PARAMETER :: RAD=PI/180.0_sp
INTEGER(I4B) :: i
REAL(SP) :: am,as,c,t,t2,xtra
c=n+nph/4.0_sp                          This is how we comment an individual line.
t=c/1236.85_sp
t2=t**2
as=359.2242_sp+29.105356_sp*c           You aren't really intended to understand this al-
am=306.0253_sp+385.816918_sp*c+0.010730_sp*t2       gorithm, but it does work!
jd=2415020+28*n+7*nph
xtra=0.75933_sp+1.53058868_sp*c+(1.178e-4_sp-1.55e-7_sp*t)*t2
select case(nph)
    case(0,2)
        xtra=xtra+(0.1734_sp-3.93e-4_sp*t)*sin(RAD*as)-0.4068_sp*sin(RAD*am)
    case(1,3)
        xtra=xtra+(0.1721_sp-4.0e-4_sp*t)*sin(RAD*as)-0.6280_sp*sin(RAD*am)
    case default
        call nrerror('flmoon: nph is unknown')      This is how we will indicate error
end select                                              conditions.
i=int(merge(xtra,xtra-1.0_sp, xtra >= 0.0))
jd=jd+i
frac=xtra-i
END SUBROUTINE flmoon
```

f90 `select case(nph)...case(0,2)...end select` Fortran 90 includes a `case` construction that executes at most one of several blocks of code, depending on the value of an integer, logical, or character expression. Ideally, the `case` construction will execute more efficiently than a long sequence of cascaded `if...else if...else if...` constructions. `C` programmers should note that the Fortran 90 construction, perhaps mercifully, does not have `C`'s "drop-through" feature.

`merge(xtra,xtra-1.0_sp, xtra >= 0.0)` The `merge` construction in Fortran 90, while intended primarily for use with vector arguments, is also a convenient way of generating conditional scalar expressions, that is, expressions with one value, or another, depending on the result of a logical test.

When the arguments of a `merge` are vectors, parallelization by the compiler is straightforward as an array parallel operation (see p. 964). Less obvious is how the scalar case, as above, is handled. For small-scale parallel (SSP) machines, the natural gain is via speculative evaluation of both of the first two arguments simultaneously with evaluation of the test.

A good compiler should not penalize a scalar machine for use of either the scalar or vector `merge` construction. The Fortran 90 standard states that "it is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise." Therefore, for each test on a scalar machine, only one or the other of the first two argument components need be evaluated.

⋆ ⋆ ⋆

```
FUNCTION julday(mm,id,iyyy)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: mm,id,iyyy
INTEGER(I4B) :: julday
    In this routine julday returns the Julian Day Number that begins at noon of the calendar
    date specified by month mm, day id, and year iyyy, all integer variables. Positive year
    signifies A.D.; negative, B.C. Remember that the year after 1 B.C. was 1 A.D.
INTEGER(I4B), PARAMETER :: IGREG=15+31*(10+12*1582)      Gregorian Calendar adopted
INTEGER(I4B) :: ja,jm,jy                                        Oct. 15, 1582.
jy=iyyy
if (jy == 0) call nrerror('julday: there is no year zero')
if (jy < 0) jy=jy+1
if (mm > 2) then                                  Here is an example of a block IF-structure.
    jm=mm+1
else
    jy=jy-1
    jm=mm+13
end if
julday=int(365.25_sp*jy)+int(30.6001_sp*jm)+id+1720995
if (id+31*(mm+12*iyyy) >= IGREG) then          Test whether to change to Gregorian Cal-
    ja=int(0.01_sp*jy)                                endar.
    julday=julday+2-ja+int(0.25_sp*ja)
end if
END FUNCTION julday
```

⋆ ⋆ ⋆

```
PROGRAM badluk
USE nrtype
USE nr, ONLY : flmoon,julday
IMPLICIT NONE
INTEGER(I4B) :: ic,icon,idwk,ifrac,im,iyyy,jd,jday,n
INTEGER(I4B) :: iybeg=1900,iyend=2000      The range of dates to be searched.
REAL(SP) :: frac
REAL(SP), PARAMETER :: TIMZON=-5.0_sp/24.0_sp
  Time zone −5 is Eastern Standard Time.
write (*,'(1x,a,i5,a,i5)') 'Full moons on Friday the 13th from',&
    iybeg,' to',iyend
do iyyy=iybeg,iyend                            Loop over each year,
    do im=1,12                                 and each month.
        jday=julday(im,13,iyyy)                Is the 13th a Friday?
        idwk=mod(jday+1,7)
```

```
        if (idwk == 5) then
            n=12.37_sp*(iyyy-1900+(im-0.5_sp)/12.0_sp)
              This value n is a first approximation to how many full moons have occurred
              since 1900. We will feed it into the phase routine and adjust it up or down until
              we determine that our desired 13th was or was not a full moon. The variable
              icon signals the direction of adjustment.
            icon=0
            do
                call flmoon(n,2,jd,frac)        Get date of full moon n.
                ifrac=nint(24.0_sp*(frac+TIMZON))     Convert to hours in correct time
                if (ifrac < 0) then                        zone.
                    jd=jd-1                 Convert from Julian Days beginning at noon
                    ifrac=ifrac+24              to civil days beginning at midnight.
                end if
                if (ifrac > 12) then
                    jd=jd+1
                    ifrac=ifrac-12
                else
                    ifrac=ifrac+12
                end if
                if (jd == jday) then        Did we hit our target day?
                    write (*,'(/1x,i2,a,i2,a,i4)') im,'/',13,'/',iyyy
                    write (*,'(1x,a,i2,a)') 'Full moon ',ifrac,&
                        ' hrs after midnight (EST).'
                      Don't worry if you are unfamiliar with FORTRAN's esoteric input/output
                      statements; very few programs in this book do any input/output.
                    exit                        Part of the break-structure, case of a match.
                else                            Didn't hit it.
                    ic=isign(1,jday-jd)
                    if (ic == -icon) exit       Another break, case of no match.
                    icon=ic
                    n=n+ic
                end if
            end do
        end if
    end do
end do
END PROGRAM badluk
```

f90 `...IGREG=15+31*(10+12*1582)` (in julday), `...TIMZON=-5.0_sp/24.0_sp` (in badluk) These are two examples of initialization expressions for "named constants" (that is, PARAMETERs). Because the initialization expressions will generally be evaluated at compile time, Fortran 90 puts some restrictions on what kinds of intrinsic functions they can contain. Although the evaluation of a real expression like `-5.0_sp/24.0_sp` *ought* to give identical results at compile time and at execution time, all the way down to the least significant bit, in our opinion the conservative programmer shouldn't count on strict identity at the level of floating-point roundoff error. (In the special case of *cross*-compilers, such roundoff-level discrepancies between compile time and run time are almost inevitable.)

⋆ ⋆ ⋆

```
SUBROUTINE caldat(julian,mm,id,iyyy)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: julian
INTEGER(I4B), INTENT(OUT) :: mm,id,iyyy
```

Inverse of the function julday given above. Here julian is input as a Julian Day Number, and the routine outputs mm,id, and iyyy as the month, day, and year on which the specified Julian Day started at noon.

```
INTEGER(I4B) :: ja,jalpha,jb,jc,jd,je
INTEGER(I4B), PARAMETER :: IGREG=2299161
if (julian >= IGREG) then                Cross-over to Gregorian Calendar produces this
    jalpha=int(((julian-1867216)-0.25_sp)/36524.25_sp)      correction.
    ja=julian+1+jalpha-int(0.25_sp*jalpha)
else if (julian < 0) then                Make day number positive by adding integer num-
    ja=julian+36525*(1-julian/36525)       ber of Julian centuries, then subtract them
else                                       off at the end.
    ja=julian
end if
jb=ja+1524
jc=int(6680.0_sp+((jb-2439870)-122.1_sp)/365.25_sp)
jd=365*jc+int(0.25_sp*jc)
je=int((jb-jd)/30.6001_sp)
id=jb-jd-int(30.6001_sp*je)
mm=je-1
if (mm > 12) mm=mm-12
iyyy=jc-4715
if (mm > 2) iyyy=iyyy-1
if (iyyy <= 0) iyyy=iyyy-1
if (julian < 0) iyyy=iyyy-100*(1-julian/36525)
END SUBROUTINE caldat
```

C1. Listing of Utility Modules (nrtype and nrutil)

C1.1 Numerical Recipes Types (nrtype)

The file supplied as `nrtype.f90` contains a single module named `nrtype`, which in turn contains definitions for a number of named constants (that is, `PARAMETER`s), and a couple of elementary derived data types used by the sparse matrix routines in this book. Of the named constants, by far the most important are those that define the `KIND` types of virtually all the variables used in this book: `I4B`, `I2B`, and `I1B` for integer variables, `SP` and `DP` for real variables (and `SPC` and `DPC` for the corresponding complex cases), and `LGT` for the default logical type.

MODULE nrtype

Symbolic names for kind types of 4-, 2-, and 1-byte integers:

```
INTEGER, PARAMETER :: I4B = SELECTED_INT_KIND(9)
INTEGER, PARAMETER :: I2B = SELECTED_INT_KIND(4)
INTEGER, PARAMETER :: I1B = SELECTED_INT_KIND(2)
```

Symbolic names for kind types of single- and double-precision reals:

```
INTEGER, PARAMETER :: SP = KIND(1.0)
INTEGER, PARAMETER :: DP = KIND(1.0D0)
```

Symbolic names for kind types of single- and double-precision complex:

```
INTEGER, PARAMETER :: SPC = KIND((1.0,1.0))
INTEGER, PARAMETER :: DPC = KIND((1.0D0,1.0D0))
```

Symbolic name for kind type of default logical:

```
INTEGER, PARAMETER :: LGT = KIND(.true.)
```

Frequently used mathematical constants (with precision to spare):

```
REAL(SP), PARAMETER :: PI=3.141592653589793238462643383279502884197_sp
REAL(SP), PARAMETER :: PIO2=1.570796326794896619231321691639751442098 58_sp
REAL(SP), PARAMETER :: TWOPI=6.283185307179586476925286766559005768394_sp
REAL(SP), PARAMETER :: SQRT2=1.414213562373095048801688724209698078569 67_sp
REAL(SP), PARAMETER :: EULER=0.577215664901532860606512090082402431042 2_sp
REAL(DP), PARAMETER :: PI_D=3.141592653589793238462643383279502884197_dp
REAL(DP), PARAMETER :: PIO2_D=1.570796326794896619231321691639751442098 58_dp
REAL(DP), PARAMETER :: TWOPI_D=6.283185307179586476925286766559005768394_dp
```

Derived data types for sparse matrices, single and double precision (see use in Chapter B2):

```
TYPE sprs2_sp
    INTEGER(I4B) :: n,len
    REAL(SP), DIMENSION(:), POINTER :: val
    INTEGER(I4B), DIMENSION(:), POINTER :: irow
    INTEGER(I4B), DIMENSION(:), POINTER :: jcol
END TYPE sprs2_sp
TYPE sprs2_dp
    INTEGER(I4B) :: n,len
    REAL(DP), DIMENSION(:), POINTER :: val
```

```
    INTEGER(I4B), DIMENSION(:), POINTER :: irow
    INTEGER(I4B), DIMENSION(:), POINTER :: jcol
END TYPE sprs2_dp
END MODULE nrtype
```

About Converting to Higher Precision

You might hope that changing all the Numerical Recipes routines from single precision to double precision would be as simple as redefining the values of SP and DP in nrtype. Well ... not quite.

Converting algorithms to a higher precision is not a purely mechanical task because of the distinction between "roundoff error" and "truncation error." (Please see Volume 1, §1.2, if you are not familiar with these concepts.) While increasing the precision implied by the kind values SP and DP will indeed reduce a routine's roundoff error, it will not reduce any truncation error that may be intrinsic to the algorithm. Sometimes, a routine contains "accuracy parameters" that can be adjusted to reduce the truncation error to the new, desired level. In other cases, however, the truncation error cannot be so easily reduced; then, a whole new algorithm is needed. Clearly such new algorithms are beyond the scope of a simple mechanical "conversion."

If, despite these cautionary words, you want to proceed with converting some routines to a higher precision, here are some hints:

If your machine has a kind type that is distinct from, and has equal or greater precision than, the kind type that we use for DP, then, in nrtype, you can simply redefine DP to this new highest precision and redefine SP to what was previously DP. For example, DEC machines usually have a "quadruple precision" real type available, which can be used in this way. You should not need to make any further edits of nrtype or nrutil.

If, on the other hand, the kind type that we already use for DP is the highest precision available, then you must leave DP defined as it is, and redefine SP in nrtype to be this same kind type. Now, however, you will also have to edit nrutil, because some overloaded routines that were previously distinguishable (by the different kind types) will now be seen by the compiler as indistinguishable — and it will object strenuously. Simply delete all the "_dp" function names from the list of overloaded procedures (i.e., from the MODULE PROCEDURE statements). Note that it is not necessary to delete the routines from the MODULE itself. Similarly, in the interface file nr.f90 you must delete the "_dp" interfaces, *except* for the sprs... routines. (Since they have TYPE(sprs2_dp) or TYPE(sprs2_sp), they are treated as distinct even though they have functionally equivalent kind types.)

Finally, the following table gives some suggestions for changing the accuracy parameters, or constants, in some of the routines. Please note that this table is not necessarily complete, and that higher-precision performance is not guaranteed for all the routines, *even if* you make all the changes indicated. The above edits, and these suggestions, do, however, work in the majority of cases.

In routine...	*change...*	*to...*
beschb	NUSE1=5,NUSE2=5	NUSE1=7,NUSE2=8
bessi	IACC=40	IACC=200
bessik	EPS=1.0e-10_dp	EPS=epsilon(x)
bessj	IACC=40	IACC=160
bessjy	EPS=1.0e-10_dp	EPS=epsilon(x)
broydn	TOLF=1.0e-4_sp	TOLF=1.0e-8_sp
	TOLMIN=1.0e-6_sp	TOLMIN=1.0e-12_sp
fdjac	EPS=1.0e-4_sp	EPS=1.0e-8_sp
frprmn	EPS=1.0e-10_sp	EPS=1.0e-18_sp
gauher	EPS=3.0e-13_dp	EPS=1.0e-14_dp
gaujac	EPS=3.0e-14_dp	EPS=1.0e-14_dp
gaulag	EPS=3.0e-13_dp	EPS=1.0e-14_dp
gauleg	EPS=3.0e-14_dp	EPS=1.0e-14_dp
hypgeo	EPS=1.0e-6_sp	EPS=1.0e-14_sp
linmin	TOL=1.0e-4_sp	TOL=1.0e-8_sp
newt	TOLF=1.0e-4_sp	TOLF=1.0e-8_sp
	TOLMIN=1.0e-6_sp	TOLMIN=1.0e-12_sp
probks	EPS1=0.001_sp	EPS1=1.0e-6_sp
	EPS2=1.0e-8_sp	EPS2=1.0e-16_sp
qromb	EPS=1.0e-6_sp	EPS=1.0e-10_sp
qromo	EPS=1.0e-6_sp	EPS=1.0e-10_sp
qroot	TINY=1.0e-6_sp	TINY=1.0e-14_sp
qsimp	EPS=1.0e-6_sp	EPS=1.0e-10_sp
qtrap	EPS=1.0e-6_sp	EPS=1.0e-10_sp
rc	ERRTOL=0.04_sp	ERRTOL=0.0012_sp
rd	ERRTOL=0.05_sp	ERRTOL=0.0015_sp
rf	ERRTOL=0.08_sp	ERRTOL=0.0025_sp
rj	ERRTOL=0.05_sp	ERRTOL=0.0015_sp
sfroid	conv=5.0e-6_sp	conv=1.0e-14_sp
shoot	EPS=1.0e-6_sp	EPS=1.0e-14_sp
shootf	EPS=1.0e-6_sp	EPS=1.0e-14_sp
simplx	EPS=1.0e-6_sp	EPS=1.0e-14_sp
sncndn	CA=0.0003_sp	CA=1.0e-8_sp
sor	EPS=1.0e-5_dp	EPS=1.0e-13_dp
sphfpt	DXX=1.0e-4_sp	DXX=1.0e-8_sp
sphoot	dx=1.0e-4_sp	dx=1.0e-8_sp
svdfit	TOL=1.0e-5_sp	TOL=1.0e-13_sp
zroots	EPS=1.0e-6_sp	EPS=1.0e-14_sp

C1.2 Numerical Recipes Utilities (nrutil)

The file supplied as `nrutil.f90` contains a single module named `nrutil`, which contains specific implementations for all the Numerical Recipes utility functions described in detail in Chapter 23.

The specific implementations given are something of a compromise between demonstrating parallel techniques (when they can be achieved in Fortran 90) and running efficiently on conventional, serial machines. The parameters at the beginning of the module (names beginning with `NPAR_`) are typically related to array lengths *below which* the implementations revert to serial operations. On a purely serial machine, these can be set to large values to suppress many parallel constructions.

The length and repetitiveness of the `nrutil.f90` file stems in large part from its extensive use of overloading. Indeed, the file would be even longer if we overloaded versions for all the applicable data types that each utility could, in principle, instantiate. The descriptions in Chapter 23 detail both the full set of intended data types and shapes for each routine, and also the types and shapes actually here implemented (which can also be gleaned by examining the file). The intended result of all this overloading is, in essence, to give the utility routines the desirable properties of many of the Fortran 90 intrinsic functions, namely, to be both *generic* (apply to many data types) and *elemental* (apply element-by-element to arbitrary shapes). Fortran 95's provision of user-defined elemental functions will reduce the multiplicity of overloading in some of our routines; unfortunately the necessity to overload for multiple data types will still be present.

Finally, it is worth reemphasizing the following point, already made in Chapter 23: The purpose of the `nrutil` utilities is to remove from the Numerical Recipes programs just those programming tasks and "idioms" whose efficient implementation is *most* hardware and compiler dependent, so as to allow for specific, efficient implementations on different machines. One should therefore not expect the utmost in efficiency from the general purpose, one-size-fits-all, implementation listed here.

Correspondingly, we would encourage the incorporation of efficient `nrutil` implementations, and/or comparable capabilities under different names, with as broad as possible a set of overloaded data types, in libraries associated with specific compilers or machines. In support of this goal, we have specifically put this Appendix C1, and the files `nrtype.f90` and `nrutil.f90`, into the public domain.

```
MODULE nrutil
  TABLE OF CONTENTS OF THE NRUTIL MODULE:
    routines that move data:
      array_copy, swap, reallocate
    routines returning a location as an integer value
      ifirstloc, imaxloc, iminloc
    routines for argument checking and error handling:
      assert, assert_eq, nrerror
    routines relating to polynomials and recurrences
      arth, geop, cumsum, cumprod, poly, polyterm,
      zroots_unity
    routines for "outer" operations on vectors
      outerand, outersum, outerdiff, outerprod, outerdiv
    routines for scatter-with-combine
      scatter_add, scatter_max
    routines for skew operations on matrices
      diagadd, diagmult, get_diag, put_diag,
```

unit_matrix, lower_triangle, upper_triangle
miscellaneous routines
vabs

```
USE nrtype
```

Parameters for crossover from serial to parallel algorithms (these are used only within this nrutil module):

```
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NPAR_ARTH=16,NPAR2_ARTH=8
INTEGER(I4B), PARAMETER :: NPAR_GEOP=4,NPAR2_GEOP=2
INTEGER(I4B), PARAMETER :: NPAR_CUMSUM=16
INTEGER(I4B), PARAMETER :: NPAR_CUMPROD=8
INTEGER(I4B), PARAMETER :: NPAR_POLY=8
INTEGER(I4B), PARAMETER :: NPAR_POLYTERM=8
```

Each NPAR2 must be $\leq$ the corresponding NPAR.

Next, generic interfaces for routines with overloaded versions. Naming conventions for appended codes in the names of overloaded routines are as follows: r=real, d=double precision, i=integer, c=complex, z=double-precision complex, h=character, l=logical. Any of r,d,i,c,z,h,l may be followed by v=vector or m=matrix (v,m suffixes are used only when needed to resolve ambiguities).

Routines that move data:

```
INTERFACE array_copy
    MODULE PROCEDURE array_copy_r, array_copy_d, array_copy_i
END INTERFACE
INTERFACE swap
    MODULE PROCEDURE swap_i,swap_r,swap_rv,swap_c, &
        swap_cv,swap_cm,swap_z,swap_zv,swap_zm, &
        masked_swap_rs,masked_swap_rv,masked_swap_rm
END INTERFACE
INTERFACE reallocate
    MODULE PROCEDURE reallocate_rv,reallocate_rm,&
        reallocate_iv,reallocate_im,reallocate_hv
END INTERFACE
```

Routines returning a location as an integer value (ifirstloc, iminloc are not currently overloaded and so do not have a generic interface here):

```
INTERFACE imaxloc
    MODULE PROCEDURE imaxloc_r,imaxloc_i
END INTERFACE
```

Routines for argument checking and error handling (nrerror is not currently overloaded):

```
INTERFACE assert
    MODULE PROCEDURE assert1,assert2,assert3,assert4,assert_v
END INTERFACE
INTERFACE assert_eq
    MODULE PROCEDURE assert_eq2,assert_eq3,assert_eq4,assert_eqn
END INTERFACE
```

Routines relating to polynomials and recurrences (cumprod, zroots_unity are not currently overloaded):

```
INTERFACE arth
    MODULE PROCEDURE arth_r, arth_d, arth_i
END INTERFACE
INTERFACE geop
    MODULE PROCEDURE geop_r, geop_d, geop_i, geop_c, geop_dv
END INTERFACE
INTERFACE cumsum
    MODULE PROCEDURE cumsum_r,cumsum_i
END INTERFACE
INTERFACE poly
    MODULE PROCEDURE poly_rr,poly_rrv,poly_dd,poly_ddv,&
        poly_rc,poly_cc,poly_msk_rrv,poly_msk_ddv
END INTERFACE
INTERFACE poly_term
    MODULE PROCEDURE poly_term_rr,poly_term_cc
END INTERFACE
```

Routines for "outer" operations on vectors (outerand, outersum, outerdiv are not currently overloaded):

```
INTERFACE outerprod
```

```
    MODULE PROCEDURE outerprod_r,outerprod_d
END INTERFACE
INTERFACE outerdiff
    MODULE PROCEDURE outerdiff_r,outerdiff_d,outerdiff_i
END INTERFACE
```
Routines for scatter-with-combine, scatter_add, scatter_max:
```
INTERFACE scatter_add
    MODULE PROCEDURE scatter_add_r,scatter_add_d
END INTERFACE
INTERFACE scatter_max
    MODULE PROCEDURE scatter_max_r,scatter_max_d
END INTERFACE
```
Routines for skew operations on matrices (unit_matrix, lower_triangle, upper_triangle not currently overloaded):
```
INTERFACE diagadd
    MODULE PROCEDURE diagadd_rv,diagadd_r
END INTERFACE
INTERFACE diagmult
    MODULE PROCEDURE diagmult_rv,diagmult_r
END INTERFACE
INTERFACE get_diag
    MODULE PROCEDURE get_diag_rv, get_diag_dv
END INTERFACE
INTERFACE put_diag
    MODULE PROCEDURE put_diag_rv, put_diag_r
END INTERFACE
```
Other routines (vabs is not currently overloaded):
```
CONTAINS
```
Routines that move data:
```
SUBROUTINE array_copy_r(src,dest,n_copied,n_not_copied)
```
Copy array where size of source not known in advance.
```
REAL(SP), DIMENSION(:), INTENT(IN) :: src
REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
END SUBROUTINE array_copy_r

SUBROUTINE array_copy_d(src,dest,n_copied,n_not_copied)
REAL(DP), DIMENSION(:), INTENT(IN) :: src
REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
END SUBROUTINE array_copy_d

SUBROUTINE array_copy_i(src,dest,n_copied,n_not_copied)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: src
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
END SUBROUTINE array_copy_i

SUBROUTINE swap_i(a,b)
```
Swap the contents of a and b.
```
INTEGER(I4B), INTENT(INOUT) :: a,b
INTEGER(I4B) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_i
```

```
SUBROUTINE swap_r(a,b)
REAL(SP), INTENT(INOUT) :: a,b
REAL(SP) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_r
SUBROUTINE swap_rv(a,b)
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
REAL(SP), DIMENSION(SIZE(a)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_rv
SUBROUTINE swap_c(a,b)
COMPLEX(SPC), INTENT(INOUT) :: a,b
COMPLEX(SPC) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_c
SUBROUTINE swap_cv(a,b)
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: a,b
COMPLEX(SPC), DIMENSION(SIZE(a)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_cv
SUBROUTINE swap_cm(a,b)
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: a,b
COMPLEX(SPC), DIMENSION(size(a,1),size(a,2)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_cm
SUBROUTINE swap_z(a,b)
COMPLEX(DPC), INTENT(INOUT) :: a,b
COMPLEX(DPC) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_z
SUBROUTINE swap_zv(a,b)
COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: a,b
COMPLEX(DPC), DIMENSION(SIZE(a)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_zv
SUBROUTINE swap_zm(a,b)
COMPLEX(DPC), DIMENSION(:,:), INTENT(INOUT) :: a,b
COMPLEX(DPC), DIMENSION(size(a,1),size(a,2)) :: dum
dum=a
a=b
b=dum
END SUBROUTINE swap_zm
SUBROUTINE masked_swap_rs(a,b,mask)
REAL(SP), INTENT(INOUT) :: a,b
LOGICAL(LGT), INTENT(IN) :: mask
REAL(SP) :: swp
```

```
if (mask) then
    swp=a
    a=b
    b=swp
end if
END SUBROUTINE masked_swap_rs

SUBROUTINE masked_swap_rv(a,b,mask)
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(a)) :: swp
where (mask)
    swp=a
    a=b
    b=swp
end where
END SUBROUTINE masked_swap_rv

SUBROUTINE masked_swap_rm(a,b,mask)
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
LOGICAL(LGT), DIMENSION(:,:), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(a,1),size(a,2)) :: swp
where (mask)
    swp=a
    a=b
    b=swp
end where
END SUBROUTINE masked_swap_rm

FUNCTION reallocate_rv(p,n)
    Reallocate a pointer to a new size, preserving its previous contents.
REAL(SP), DIMENSION(:), POINTER :: p, reallocate_rv
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B) :: nold,ierr
allocate(reallocate_rv(n),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_rv: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate_rv(1:min(nold,n))=p(1:min(nold,n))
deallocate(p)
END FUNCTION reallocate_rv

FUNCTION reallocate_iv(p,n)
INTEGER(I4B), DIMENSION(:), POINTER :: p, reallocate_iv
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B) :: nold,ierr
allocate(reallocate_iv(n),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_iv: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate_iv(1:min(nold,n))=p(1:min(nold,n))
deallocate(p)
END FUNCTION reallocate_iv

FUNCTION reallocate_hv(p,n)
CHARACTER(1), DIMENSION(:), POINTER :: p, reallocate_hv
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B) :: nold,ierr
allocate(reallocate_hv(n),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_hv: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate_hv(1:min(nold,n))=p(1:min(nold,n))
```

```
deallocate(p)
END FUNCTION reallocate_hv

FUNCTION reallocate_rm(p,n,m)
REAL(SP), DIMENSION(:,:), POINTER :: p, reallocate_rm
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B) :: nold,mold,ierr
allocate(reallocate_rm(n,m),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_rm: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p,1)
mold=size(p,2)
reallocate_rm(1:min(nold,n),1:min(mold,m))=&
    p(1:min(nold,n),1:min(mold,m))
deallocate(p)
END FUNCTION reallocate_rm

FUNCTION reallocate_im(p,n,m)
INTEGER(I4B), DIMENSION(:,:), POINTER :: p, reallocate_im
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B) :: nold,mold,ierr
allocate(reallocate_im(n,m),stat=ierr)
if (ierr /= 0) call &
    nrerror('reallocate_im: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p,1)
mold=size(p,2)
reallocate_im(1:min(nold,n),1:min(mold,m))=&
    p(1:min(nold,n),1:min(mold,m))
deallocate(p)
END FUNCTION reallocate_im
```

Routines returning a location as an integer value:

```
FUNCTION ifirstloc(mask)
```

Index of first occurrence of .true. in a logical vector.

```
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
INTEGER(I4B) :: ifirstloc
INTEGER(I4B), DIMENSION(1) :: loc
loc=maxloc(merge(1,0,mask))
ifirstloc=loc(1)
if (.not. mask(ifirstloc)) ifirstloc=size(mask)+1
END FUNCTION ifirstloc

FUNCTION imaxloc_r(arr)
```

Index of maxloc on an array.

```
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B) :: imaxloc_r
INTEGER(I4B), DIMENSION(1) :: imax
imax=maxloc(arr(:))
imaxloc_r=imax(1)
END FUNCTION imaxloc_r

FUNCTION imaxloc_i(iarr)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
INTEGER(I4B), DIMENSION(1) :: imax
INTEGER(I4B) :: imaxloc_i
imax=maxloc(iarr(:))
imaxloc_i=imax(1)
END FUNCTION imaxloc_i

FUNCTION iminloc(arr)
```

Index of minloc on an array.

```
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(1) :: imin
INTEGER(I4B) :: iminloc
imin=minloc(arr(:))
```

```
iminloc=imin(1)
END FUNCTION iminloc
```

Routines for argument checking and error handling:

```
SUBROUTINE assert1(n1,string)
```

Report and die if any logical is false (used for arg range checking).

```
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1
if (.not. n1) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert1'
end if
END SUBROUTINE assert1

SUBROUTINE assert2(n1,n2,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2
if (.not. (n1 .and. n2)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert2'
end if
END SUBROUTINE assert2

SUBROUTINE assert3(n1,n2,n3,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2,n3
if (.not. (n1 .and. n2 .and. n3)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert3'
end if
END SUBROUTINE assert3

SUBROUTINE assert4(n1,n2,n3,n4,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2,n3,n4
if (.not. (n1 .and. n2 .and. n3 .and. n4)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert4'
end if
END SUBROUTINE assert4

SUBROUTINE assert_v(n,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, DIMENSION(:), INTENT(IN) :: n
if (.not. all(n)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
        string
    STOP 'program terminated by assert_v'
end if
END SUBROUTINE assert_v

FUNCTION assert_eq2(n1,n2,string)
```

Report and die if integers not all equal (used for size checking).

```
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2
INTEGER :: assert_eq2
if (n1 == n2) then
    assert_eq2=n1
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eq2'
end if
```

```
END FUNCTION assert_eq2

FUNCTION assert_eq3(n1,n2,n3,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3
INTEGER :: assert_eq3
if (n1 == n2 .and. n2 == n3) then
    assert_eq3=n1
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eq3'
end if
END FUNCTION assert_eq3

FUNCTION assert_eq4(n1,n2,n3,n4,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3,n4
INTEGER :: assert_eq4
if (n1 == n2 .and. n2 == n3 .and. n3 == n4) then
    assert_eq4=n1
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eq4'
end if
END FUNCTION assert_eq4

FUNCTION assert_eqn(nn,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, DIMENSION(:), INTENT(IN) :: nn
INTEGER :: assert_eqn
if (all(nn(2:) == nn(1))) then
    assert_eqn=nn(1)
else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
        string
    STOP 'program terminated by assert_eqn'
end if
END FUNCTION assert_eqn

SUBROUTINE nrerror(string)
   Report a message, then die.
CHARACTER(LEN=*), INTENT(IN) :: string
write (*,*) 'nrerror: ',string
STOP 'program terminated by nrerror'
END SUBROUTINE nrerror
```

Routines relating to polynomials and recurrences:

```
FUNCTION arth_r(first,increment,n)
```

Array function returning an arithmetic progression.

```
REAL(SP), INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: arth_r
INTEGER(I4B) :: k,k2
REAL(SP) :: temp
if (n > 0) arth_r(1)=first
if (n <= NPAR_ARTH) then
    do k=2,n
        arth_r(k)=arth_r(k-1)+increment
    end do
else
    do k=2,NPAR2_ARTH
        arth_r(k)=arth_r(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
```

```
    do
        if (k >= n) exit
        k2=k+k
        arth_r(k+1:min(k2,n))=temp+arth_r(1:min(k,n-k))
        temp=temp+temp
        k=k2
    end do
end if
END FUNCTION arth_r

FUNCTION arth_d(first,increment,n)
REAL(DP), INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(n) :: arth_d
INTEGER(I4B) :: k,k2
REAL(DP) :: temp
if (n > 0) arth_d(1)=first
if (n <= NPAR_ARTH) then
    do k=2,n
        arth_d(k)=arth_d(k-1)+increment
    end do
else
    do k=2,NPAR2_ARTH
        arth_d(k)=arth_d(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
    do
        if (k >= n) exit
        k2=k+k
        arth_d(k+1:min(k2,n))=temp+arth_d(1:min(k,n-k))
        temp=temp+temp
        k=k2
    end do
end if
END FUNCTION arth_d

FUNCTION arth_i(first,increment,n)
INTEGER(I4B), INTENT(IN) :: first,increment,n
INTEGER(I4B), DIMENSION(n) :: arth_i
INTEGER(I4B) :: k,k2,temp
if (n > 0) arth_i(1)=first
if (n <= NPAR_ARTH) then
    do k=2,n
        arth_i(k)=arth_i(k-1)+increment
    end do
else
    do k=2,NPAR2_ARTH
        arth_i(k)=arth_i(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
    do
        if (k >= n) exit
        k2=k+k
        arth_i(k+1:min(k2,n))=temp+arth_i(1:min(k,n-k))
        temp=temp+temp
        k=k2
    end do
end if
END FUNCTION arth_i

FUNCTION geop_r(first,factor,n)
    Array function returning a geometric progression.
REAL(SP), INTENT(IN) :: first,factor
```

```
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: geop_r
INTEGER(I4B) :: k,k2
REAL(SP) :: temp
if (n > 0) geop_r(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_r(k)=geop_r(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_r(k)=geop_r(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_r(k+1:min(k2,n))=temp*geop_r(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_r
FUNCTION geop_d(first,factor,n)
REAL(DP), INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(n) :: geop_d
INTEGER(I4B) :: k,k2
REAL(DP) :: temp
if (n > 0) geop_d(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_d(k)=geop_d(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_d(k)=geop_d(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_d(k+1:min(k2,n))=temp*geop_d(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_d
FUNCTION geop_i(first,factor,n)
INTEGER(I4B), INTENT(IN) :: first,factor,n
INTEGER(I4B), DIMENSION(n) :: geop_i
INTEGER(I4B) :: k,k2,temp
if (n > 0) geop_i(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_i(k)=geop_i(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_i(k)=geop_i(k-1)*factor
    end do
```

```
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_i(k+1:min(k2,n))=temp*geop_i(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_i

FUNCTION geop_c(first,factor,n)
COMPLEX(SP), INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
COMPLEX(SP), DIMENSION(n) :: geop_c
INTEGER(I4B) :: k,k2
COMPLEX(SP) :: temp
if (n > 0) geop_c(1)=first
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_c(k)=geop_c(k-1)*factor
    end do
else
    do k=2,NPAR2_GEOP
        geop_c(k)=geop_c(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_c(k+1:min(k2,n))=temp*geop_c(1:min(k,n-k))
        temp=temp*temp
        k=k2
    end do
end if
END FUNCTION geop_c

FUNCTION geop_dv(first,factor,n)
REAL(DP), DIMENSION(:), INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
REAL(DP), DIMENSION(size(first),n) :: geop_dv
INTEGER(I4B) :: k,k2
REAL(DP), DIMENSION(size(first)) :: temp
if (n > 0) geop_dv(:,1)=first(:)
if (n <= NPAR_GEOP) then
    do k=2,n
        geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
else
    do k=2,NPAR2_GEOP
        geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
        if (k >= n) exit
        k2=k+k
        geop_dv(:,k+1:min(k2,n))=geop_dv(:,1:min(k,n-k))*&
            spread(temp,2,size(geop_dv(:,1:min(k,n-k)),2))
        temp=temp*temp
        k=k2
    end do
end if
```

```
END FUNCTION geop_dv

RECURSIVE FUNCTION cumsum_r(arr,seed) RESULT(ans)
    Cumulative sum on an array, with optional additive seed.
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP), OPTIONAL, INTENT(IN) :: seed
REAL(SP), DIMENSION(size(arr)) :: ans
INTEGER(I4B) :: n,j
REAL(SP) :: sd
n=size(arr)
if (n == 0_i4b) RETURN
sd=0.0_sp
if (present(seed)) sd=seed
ans(1)=arr(1)+sd
if (n < NPAR_CUMSUM) then
    do j=2,n
        ans(j)=ans(j-1)+arr(j)
    end do
else
    ans(2:n:2)=cumsum_r(arr(2:n:2)+arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
end if
END FUNCTION cumsum_r

RECURSIVE FUNCTION cumsum_i(arr,seed) RESULT(ans)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), OPTIONAL, INTENT(IN) :: seed
INTEGER(I4B), DIMENSION(size(arr)) :: ans
INTEGER(I4B) :: n,j,sd
n=size(arr)
if (n == 0_i4b) RETURN
sd=0_i4b
if (present(seed)) sd=seed
ans(1)=arr(1)+sd
if (n < NPAR_CUMSUM) then
    do j=2,n
        ans(j)=ans(j-1)+arr(j)
    end do
else
    ans(2:n:2)=cumsum_i(arr(2:n:2)+arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
end if
END FUNCTION cumsum_i

RECURSIVE FUNCTION cumprod(arr,seed) RESULT(ans)
    Cumulative product on an array, with optional multiplicative seed.
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP), OPTIONAL, INTENT(IN) :: seed
REAL(SP), DIMENSION(size(arr)) :: ans
INTEGER(I4B) :: n,j
REAL(SP) :: sd
n=size(arr)
if (n == 0_i4b) RETURN
sd=1.0_sp
if (present(seed)) sd=seed
ans(1)=arr(1)*sd
if (n < NPAR_CUMPROD) then
    do j=2,n
        ans(j)=ans(j-1)*arr(j)
    end do
else
    ans(2:n:2)=cumprod(arr(2:n:2)*arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)*arr(3:n:2)
end if
END FUNCTION cumprod
```

```
FUNCTION poly_rr(x,coeffs)
   Polynomial evaluation.
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
REAL(SP) :: poly_rr
REAL(SP) :: pow
REAL(SP), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_rr=0.0_sp
else if (n < NPAR_POLY) then
    poly_rr=coeffs(n)
    do i=n-1,1,-1
        poly_rr=x*poly_rr+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_sp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_rr=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_rr
FUNCTION poly_dd(x,coeffs)
REAL(DP), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs
REAL(DP) :: poly_dd
REAL(DP) :: pow
REAL(DP), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_dd=0.0_dp
else if (n < NPAR_POLY) then
    poly_dd=coeffs(n)
    do i=n-1,1,-1
        poly_dd=x*poly_dd+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_dp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_dd=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_dd
```

```
FUNCTION poly_rc(x,coeffs)
COMPLEX(SPC), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
COMPLEX(SPC) :: poly_rc
COMPLEX(SPC) :: pow
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_rc=0.0_sp
else if (n < NPAR_POLY) then
    poly_rc=coeffs(n)
    do i=n-1,1,-1
        poly_rc=x*poly_rc+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_sp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_rc=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_rc

FUNCTION poly_cc(x,coeffs)
COMPLEX(SPC), INTENT(IN) :: x
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: coeffs
COMPLEX(SPC) :: poly_cc
COMPLEX(SPC) :: pow
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
INTEGER(I4B) :: i,n,nn
n=size(coeffs)
if (n <= 0) then
    poly_cc=0.0_sp
else if (n < NPAR_POLY) then
    poly_cc=coeffs(n)
    do i=n-1,1,-1
        poly_cc=x*poly_cc+coeffs(i)
    end do
else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
        vec(n+1)=0.0_sp
        nn=ishft(n+1,-1)
        vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
        if (nn == 1) exit
        pow=pow*pow
        n=nn
    end do
    poly_cc=vec(1)
    deallocate(vec)
end if
END FUNCTION poly_cc

FUNCTION poly_rrv(x,coeffs)
```

```
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs,x
REAL(SP), DIMENSION(size(x)) :: poly_rrv
INTEGER(I4B) :: i,n,m
m=size(coeffs)
n=size(x)
if (m <= 0) then
    poly_rrv=0.0_sp
else if (m < n .or. m < NPAR_POLY) then
    poly_rrv=coeffs(m)
    do i=m-1,1,-1
        poly_rrv=x*poly_rrv+coeffs(i)
    end do
else
    do i=1,n
        poly_rrv(i)=poly_rr(x(i),coeffs)
    end do
end if
END FUNCTION poly_rrv

FUNCTION poly_ddv(x,coeffs)
REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs,x
REAL(DP), DIMENSION(size(x)) :: poly_ddv
INTEGER(I4B) :: i,n,m
m=size(coeffs)
n=size(x)
if (m <= 0) then
    poly_ddv=0.0_dp
else if (m < n .or. m < NPAR_POLY) then
    poly_ddv=coeffs(m)
    do i=m-1,1,-1
        poly_ddv=x*poly_ddv+coeffs(i)
    end do
else
    do i=1,n
        poly_ddv(i)=poly_dd(x(i),coeffs)
    end do
end if
END FUNCTION poly_ddv

FUNCTION poly_msk_rrv(x,coeffs,mask)
REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs,x
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(x)) :: poly_msk_rrv
poly_msk_rrv=unpack(poly_rrv(pack(x,mask),coeffs),mask,0.0_sp)
END FUNCTION poly_msk_rrv

FUNCTION poly_msk_ddv(x,coeffs,mask)
REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs,x
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
REAL(DP), DIMENSION(size(x)) :: poly_msk_ddv
poly_msk_ddv=unpack(poly_ddv(pack(x,mask),coeffs),mask,0.0_dp)
END FUNCTION poly_msk_ddv

RECURSIVE FUNCTION poly_term_rr(a,b) RESULT(u)
    Tabulate cumulants of a polynomial.
REAL(SP), DIMENSION(:), INTENT(IN) :: a
REAL(SP), INTENT(IN) :: b
REAL(SP), DIMENSION(size(a)) :: u
INTEGER(I4B) :: n,j
n=size(a)
if (n <= 0) RETURN
u(1)=a(1)
if (n < NPAR_POLYTERM) then
    do j=2,n
        u(j)=a(j)+b*u(j-1)
    end do
```

```
else
    u(2:n:2)=poly_term_rr(a(2:n:2)+a(1:n-1:2)*b,b*b)
    u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
end if
END FUNCTION poly_term_rr

RECURSIVE FUNCTION poly_term_cc(a,b) RESULT(u)
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
COMPLEX(SPC), INTENT(IN) :: b
COMPLEX(SPC), DIMENSION(size(a)) :: u
INTEGER(I4B) :: n,j
n=size(a)
if (n <= 0) RETURN
u(1)=a(1)
if (n < NPAR_POLYTERM) then
    do j=2,n
        u(j)=a(j)+b*u(j-1)
    end do
else
    u(2:n:2)=poly_term_cc(a(2:n:2)+a(1:n-1:2)*b,b*b)
    u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
end if
END FUNCTION poly_term_cc

FUNCTION zroots_unity(n,nn)
    Complex function returning nn powers of the nth root of unity.
INTEGER(I4B), INTENT(IN) :: n,nn
COMPLEX(SPC), DIMENSION(nn) :: zroots_unity
INTEGER(I4B) :: k
REAL(SP) :: theta
zroots_unity(1)=1.0
theta=TWOPI/n
k=1
do
    if (k >= nn) exit
    zroots_unity(k+1)=cmplx(cos(k*theta),sin(k*theta),SPC)
    zroots_unity(k+2:min(2*k,nn))=zroots_unity(k+1)*&
        zroots_unity(2:min(k,nn-k))
    k=2*k
end do
END FUNCTION zroots_unity
```

Routines for "outer" operations on vectors. The order convention is: result(i,j) = first_operand(i) (op) second_operand(j).

```
FUNCTION outerprod_r(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outerprod_r
outerprod_r = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerprod_r

FUNCTION outerprod_d(a,b)
REAL(DP), DIMENSION(:), INTENT(IN) :: a,b
REAL(DP), DIMENSION(size(a),size(b)) :: outerprod_d
outerprod_d = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerprod_d

FUNCTION outerdiv(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outerdiv
outerdiv = spread(a,dim=2,ncopies=size(b)) / &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiv

FUNCTION outersum(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
```

```
REAL(SP), DIMENSION(size(a),size(b)) :: outersum
outersum = spread(a,dim=2,ncopies=size(b)) + &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outersum

FUNCTION outerdiff_r(a,b)
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a),size(b)) :: outerdiff_r
outerdiff_r = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_r

FUNCTION outerdiff_d(a,b)
REAL(DP), DIMENSION(:), INTENT(IN) :: a,b
REAL(DP), DIMENSION(size(a),size(b)) :: outerdiff_d
outerdiff_d = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_d

FUNCTION outerdiff_i(a,b)
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: a,b
INTEGER(I4B), DIMENSION(size(a),size(b)) :: outerdiff_i
outerdiff_i = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerdiff_i

FUNCTION outerand(a,b)
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: a,b
LOGICAL(LGT), DIMENSION(size(a),size(b)) :: outerand
outerand = spread(a,dim=2,ncopies=size(b)) .and. &
    spread(b,dim=1,ncopies=size(a))
END FUNCTION outerand
```

Routines for scatter-with-combine.

```
SUBROUTINE scatter_add_r(dest,source,dest_index)
REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
REAL(SP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_add_r')
m=size(dest)
do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_add_r
SUBROUTINE scatter_add_d(dest,source,dest_index)
REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
REAL(DP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_add_d')
m=size(dest)
do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
END SUBROUTINE scatter_add_d
SUBROUTINE scatter_max_r(dest,source,dest_index)
REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
REAL(SP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_max_r')
m=size(dest)
do j=1,n
    i=dest_index(j)
```

```
    if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
end do
END SUBROUTINE scatter_max_r
SUBROUTINE scatter_max_d(dest,source,dest_index)
REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
REAL(DP), DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
INTEGER(I4B) :: m,n,j,i
n=assert_eq2(size(source),size(dest_index),'scatter_max_d')
m=size(dest)
do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
end do
END SUBROUTINE scatter_max_d
```

Routines for skew operations on matrices:

```
SUBROUTINE diagadd_rv(mat,diag)
```

Adds vector or scalar diag to the diagonal of matrix mat.

```
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), DIMENSION(:), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = assert_eq2(size(diag),min(size(mat,1),size(mat,2)),'diagadd_rv')
do j=1,n
    mat(j,j)=mat(j,j)+diag(j)
end do
END SUBROUTINE diagadd_rv

SUBROUTINE diagadd_r(mat,diag)
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
    mat(j,j)=mat(j,j)+diag
end do
END SUBROUTINE diagadd_r

SUBROUTINE diagmult_rv(mat,diag)
```

Multiplies vector or scalar diag into the diagonal of matrix mat.

```
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), DIMENSION(:), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = assert_eq2(size(diag),min(size(mat,1),size(mat,2)),'diagmult_rv')
do j=1,n
    mat(j,j)=mat(j,j)*diag(j)
end do
END SUBROUTINE diagmult_rv

SUBROUTINE diagmult_r(mat,diag)
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
REAL(SP), INTENT(IN) :: diag
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
    mat(j,j)=mat(j,j)*diag
end do
END SUBROUTINE diagmult_r

FUNCTION get_diag_rv(mat)
```

Return as a vector the diagonal of matrix mat.

```
REAL(SP), DIMENSION(:,:), INTENT(IN) :: mat
REAL(SP), DIMENSION(size(mat,1)) :: get_diag_rv
INTEGER(I4B) :: j
j=assert_eq2(size(mat,1),size(mat,2),'get_diag_rv')
do j=1,size(mat,1)
    get_diag_rv(j)=mat(j,j)
```

```
end do
END FUNCTION get_diag_rv
FUNCTION get_diag_dv(mat)
REAL(DP), DIMENSION(:,:), INTENT(IN) :: mat
REAL(DP), DIMENSION(size(mat,1)) :: get_diag_dv
INTEGER(I4B) :: j
j=assert_eq2(size(mat,1),size(mat,2),'get_diag_dv')
do j=1,size(mat,1)
    get_diag_dv(j)=mat(j,j)
end do
END FUNCTION get_diag_dv
SUBROUTINE put_diag_rv(diagv,mat)
    Set the diagonal of matrix mat to the values of a vector or scalar.
REAL(SP), DIMENSION(:), INTENT(IN) :: diagv
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
INTEGER(I4B) :: j,n
n=assert_eq2(size(diagv),min(size(mat,1),size(mat,2)),'put_diag_rv')
do j=1,n
    mat(j,j)=diagv(j)
end do
END SUBROUTINE put_diag_rv
SUBROUTINE put_diag_r(scal,mat)
REAL(SP), INTENT(IN) :: scal
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
INTEGER(I4B) :: j,n
n = min(size(mat,1),size(mat,2))
do j=1,n
    mat(j,j)=scal
end do
END SUBROUTINE put_diag_r
SUBROUTINE unit_matrix(mat)
    Set the matrix mat to be a unit matrix (if it is square).
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: mat
INTEGER(I4B) :: i,n
n=min(size(mat,1),size(mat,2))
mat(:,:)=0.0_sp
do i=1,n
    mat(i,i)=1.0_sp
end do
END SUBROUTINE unit_matrix
FUNCTION upper_triangle(j,k,extra)
    Return an upper triangular logical mask.
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: upper_triangle
INTEGER(I4B) :: n
n=0
if (present(extra)) n=extra
upper_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) < n)
END FUNCTION upper_triangle
FUNCTION lower_triangle(j,k,extra)
    Return a lower triangular logical mask.
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: lower_triangle
INTEGER(I4B) :: n
n=0
if (present(extra)) n=extra
lower_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) > -n)
END FUNCTION lower_triangle
```

Other routines:

```
FUNCTION vabs(v)
    Return the length (ordinary L2 norm) of a vector.
REAL(SP), DIMENSION(:), INTENT(IN) :: v
REAL(SP) :: vabs
vabs=sqrt(dot_product(v,v))
END FUNCTION vabs
```

END MODULE nrutil

C2. Alphabetical Listing of Explicit Interfaces

The file supplied as `nr.f90` contains explicit interfaces for all the Numerical Recipes routines (except those already in the module `nrutil`). The interfaces are in alphabetical order, by the generic interface name, if one exists, or by the specific routine name if there is no generic name.

The file `nr.f90` is normally invoked via a `USE` statement within a main program or subroutine that references a Numerical Recipes routine. See §21.1 for an example.

```
MODULE nr
INTERFACE
    SUBROUTINE airy(x,ai,bi,aip,bip)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: ai,bi,aip,bip
    END SUBROUTINE airy
END INTERFACE
INTERFACE
    SUBROUTINE amebsa(p,y,pb,yb,ftol,func,iter,temptr)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: iter
    REAL(SP), INTENT(INOUT) :: yb
    REAL(SP), INTENT(IN) :: ftol,temptr
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y,pb
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE amebsa
END INTERFACE
INTERFACE
    SUBROUTINE amoeba(p,y,ftol,func,iter)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: ftol
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
```

```
    END SUBROUTINE amoeba
END INTERFACE
INTERFACE
    SUBROUTINE anneal(x,y,iorder)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    END SUBROUTINE anneal
END INTERFACE
INTERFACE
    SUBROUTINE asolve(b,x,itrnsp)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: b
    REAL(DP), DIMENSION(:), INTENT(OUT) :: x
    INTEGER(I4B), INTENT(IN) :: itrnsp
    END SUBROUTINE asolve
END INTERFACE
INTERFACE
    SUBROUTINE atimes(x,r,itrnsp)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x
    REAL(DP), DIMENSION(:), INTENT(OUT) :: r
    INTEGER(I4B), INTENT(IN) :: itrnsp
    END SUBROUTINE atimes
END INTERFACE
INTERFACE
    SUBROUTINE avevar(data,ave,var)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data
    REAL(SP), INTENT(OUT) :: ave,var
    END SUBROUTINE avevar
END INTERFACE
INTERFACE
    SUBROUTINE balanc(a)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    END SUBROUTINE balanc
END INTERFACE
INTERFACE
    SUBROUTINE banbks(a,m1,m2,al,indx,b)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: m1,m2
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,al
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
    END SUBROUTINE banbks
END INTERFACE
INTERFACE
    SUBROUTINE bandec(a,m1,m2,al,indx,d)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: m1,m2
    INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
    REAL(SP), INTENT(OUT) :: d
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: al
    END SUBROUTINE bandec
END INTERFACE
INTERFACE
    SUBROUTINE banmul(a,m1,m2,x,b)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: m1,m2
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: b
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
```

```
    END SUBROUTINE banmul
END INTERFACE
INTERFACE
    SUBROUTINE bcucof(y,y1,y2,y12,d1,d2,c)
    USE nrtype
    REAL(SP), INTENT(IN) :: d1,d2
    REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
    REAL(SP), DIMENSION(4,4), INTENT(OUT) :: c
    END SUBROUTINE bcucof
END INTERFACE
INTERFACE
    SUBROUTINE bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,&
        ansy1,ansy2)
    USE nrtype
    REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
    REAL(SP), INTENT(IN) :: x1l,x1u,x2l,x2u,x1,x2
    REAL(SP), INTENT(OUT) :: ansy,ansy1,ansy2
    END SUBROUTINE bcuint
END INTERFACE
INTERFACE beschb
    SUBROUTINE beschb_s(x,gam1,gam2,gampl,gammi)
    USE nrtype
    REAL(DP), INTENT(IN) :: x
    REAL(DP), INTENT(OUT) :: gam1,gam2,gampl,gammi
    END SUBROUTINE beschb_s

    SUBROUTINE beschb_v(x,gam1,gam2,gampl,gammi)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x
    REAL(DP), DIMENSION(:), INTENT(OUT) :: gam1,gam2,gampl,gammi
    END SUBROUTINE beschb_v
END INTERFACE
INTERFACE bessi
    FUNCTION bessi_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessi_s
    END FUNCTION bessi_s

    FUNCTION bessi_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessi_v
    END FUNCTION bessi_v
END INTERFACE
INTERFACE bessi0
    FUNCTION bessi0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessi0_s
    END FUNCTION bessi0_s

    FUNCTION bessi0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessi0_v
    END FUNCTION bessi0_v
END INTERFACE
INTERFACE bessi1
    FUNCTION bessi1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessi1_s
    END FUNCTION bessi1_s
```

```
    FUNCTION bessi1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessi1_v
    END FUNCTION bessi1_v
END INTERFACE
INTERFACE
    SUBROUTINE bessik(x,xnu,ri,rk,rip,rkp)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,xnu
    REAL(SP), INTENT(OUT) :: ri,rk,rip,rkp
    END SUBROUTINE bessik
END INTERFACE
INTERFACE bessj
    FUNCTION bessj_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessj_s
    END FUNCTION bessj_s

    FUNCTION bessj_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessj_v
    END FUNCTION bessj_v
END INTERFACE
INTERFACE bessj0
    FUNCTION bessj0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessj0_s
    END FUNCTION bessj0_s

    FUNCTION bessj0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessj0_v
    END FUNCTION bessj0_v
END INTERFACE
INTERFACE bessj1
    FUNCTION bessj1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessj1_s
    END FUNCTION bessj1_s

    FUNCTION bessj1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessj1_v
    END FUNCTION bessj1_v
END INTERFACE
INTERFACE bessjy
    SUBROUTINE bessjy_s(x,xnu,rj,ry,rjp,ryp)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,xnu
    REAL(SP), INTENT(OUT) :: rj,ry,rjp,ryp
    END SUBROUTINE bessjy_s

    SUBROUTINE bessjy_v(x,xnu,rj,ry,rjp,ryp)
    USE nrtype
    REAL(SP), INTENT(IN) :: xnu
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: rj,rjp,ry,ryp
```

```
    END SUBROUTINE bessjy_v
END INTERFACE
INTERFACE bessk
    FUNCTION bessk_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk_s
    END FUNCTION bessk_s

    FUNCTION bessk_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk_v
    END FUNCTION bessk_v
END INTERFACE
INTERFACE bessk0
    FUNCTION bessk0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk0_s
    END FUNCTION bessk0_s

    FUNCTION bessk0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk0_v
    END FUNCTION bessk0_v
END INTERFACE
INTERFACE bessk1
    FUNCTION bessk1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk1_s
    END FUNCTION bessk1_s

    FUNCTION bessk1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk1_v
    END FUNCTION bessk1_v
END INTERFACE
INTERFACE bessy
    FUNCTION bessy_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy_s
    END FUNCTION bessy_s

    FUNCTION bessy_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessy_v
    END FUNCTION bessy_v
END INTERFACE
INTERFACE bessy0
    FUNCTION bessy0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy0_s
    END FUNCTION bessy0_s

    FUNCTION bessy0_v(x)
    USE nrtype
```

```
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessy0_v
    END FUNCTION bessy0_v
END INTERFACE
INTERFACE bessy1
    FUNCTION bessy1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy1_s
    END FUNCTION bessy1_s

    FUNCTION bessy1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessy1_v
    END FUNCTION bessy1_v
END INTERFACE
INTERFACE beta
    FUNCTION beta_s(z,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: z,w
    REAL(SP) :: beta_s
    END FUNCTION beta_s

    FUNCTION beta_v(z,w)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: z,w
    REAL(SP), DIMENSION(size(z)) :: beta_v
    END FUNCTION beta_v
END INTERFACE
INTERFACE betacf
    FUNCTION betacf_s(a,b,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,x
    REAL(SP) :: betacf_s
    END FUNCTION betacf_s

    FUNCTION betacf_v(a,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
    REAL(SP), DIMENSION(size(x)) :: betacf_v
    END FUNCTION betacf_v
END INTERFACE
INTERFACE betai
    FUNCTION betai_s(a,b,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,x
    REAL(SP) :: betai_s
    END FUNCTION betai_s

    FUNCTION betai_v(a,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
    REAL(SP), DIMENSION(size(a)) :: betai_v
    END FUNCTION betai_v
END INTERFACE
INTERFACE bico
    FUNCTION bico_s(n,k)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n,k
    REAL(SP) :: bico_s
    END FUNCTION bico_s

    FUNCTION bico_v(n,k)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n,k
    REAL(SP), DIMENSION(size(n)) :: bico_v
```

```
    END FUNCTION bico_v
END INTERFACE
INTERFACE
    FUNCTION bnldev(pp,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: pp
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP) :: bnldev
    END FUNCTION bnldev
END INTERFACE
INTERFACE
    FUNCTION brent(ax,bx,cx,func,tol,xmin)
    USE nrtype
    REAL(SP), INTENT(IN) :: ax,bx,cx,tol
    REAL(SP), INTENT(OUT) :: xmin
    REAL(SP) :: brent
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION brent
END INTERFACE
INTERFACE
    SUBROUTINE broydn(x,check)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    LOGICAL(LGT), INTENT(OUT) :: check
    END SUBROUTINE broydn
END INTERFACE
INTERFACE
    SUBROUTINE bsstep(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(OUT) :: hdid,hnext
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE bsstep
END INTERFACE
INTERFACE
    SUBROUTINE caldat(julian,mm,id,iyyy)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: julian
    INTEGER(I4B), INTENT(OUT) :: mm,id,iyyy
    END SUBROUTINE caldat
END INTERFACE
INTERFACE
    FUNCTION chder(a,b,c)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(size(c)) :: chder
    END FUNCTION chder
```

```
END INTERFACE
INTERFACE chebev
    FUNCTION chebev_s(a,b,c,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,x
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP) :: chebev_s
    END FUNCTION chebev_s

    FUNCTION chebev_v(a,b,c,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: c,x
    REAL(SP), DIMENSION(size(x)) :: chebev_v
    END FUNCTION chebev_v
END INTERFACE
INTERFACE
    FUNCTION chebft(a,b,n,func)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: chebft
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION chebft
END INTERFACE
INTERFACE
    FUNCTION chebpc(c)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(size(c)) :: chebpc
    END FUNCTION chebpc
END INTERFACE
INTERFACE
    FUNCTION chint(a,b,c)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(size(c)) :: chint
    END FUNCTION chint
END INTERFACE
INTERFACE
    SUBROUTINE choldc(a,p)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: p
    END SUBROUTINE choldc
END INTERFACE
INTERFACE
    SUBROUTINE cholsl(a,p,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
    REAL(SP), DIMENSION(:), INTENT(IN) :: p,b
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    END SUBROUTINE cholsl
END INTERFACE
INTERFACE
    SUBROUTINE chsone(bins,ebins,knstrn,df,chsq,prob)
    USE nrtype
```

```
    INTEGER(I4B), INTENT(IN) :: knstrn
    REAL(SP), INTENT(OUT) :: df,chsq,prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: bins,ebins
    END SUBROUTINE chsone
END INTERFACE
INTERFACE
    SUBROUTINE chstwo(bins1,bins2,knstrn,df,chsq,prob)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: knstrn
    REAL(SP), INTENT(OUT) :: df,chsq,prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: bins1,bins2
    END SUBROUTINE chstwo
END INTERFACE
INTERFACE
    SUBROUTINE cisi(x,ci,si)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: ci,si
    END SUBROUTINE cisi
END INTERFACE
INTERFACE
    SUBROUTINE cntab1(nn,chisq,df,prob,cramrv,ccc)
    USE nrtype
    INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
    REAL(SP), INTENT(OUT) :: chisq,df,prob,cramrv,ccc
    END SUBROUTINE cntab1
END INTERFACE
INTERFACE
    SUBROUTINE cntab2(nn,h,hx,hy,hygx,hxgy,uygx,uxgy,uxy)
    USE nrtype
    INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
    REAL(SP), INTENT(OUT) :: h,hx,hy,hygx,hxgy,uygx,uxgy,uxy
    END SUBROUTINE cntab2
END INTERFACE
INTERFACE
    FUNCTION convlv(data,respns,isign)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data
    REAL(SP), DIMENSION(:), INTENT(IN) :: respns
    INTEGER(I4B), INTENT(IN) :: isign
    REAL(SP), DIMENSION(size(data)) :: convlv
    END FUNCTION convlv
END INTERFACE
INTERFACE
    FUNCTION correl(data1,data2)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    REAL(SP), DIMENSION(size(data1)) :: correl
    END FUNCTION correl
END INTERFACE
INTERFACE
    SUBROUTINE cosft1(y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    END SUBROUTINE cosft1
END INTERFACE
INTERFACE
    SUBROUTINE cosft2(y,isign)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE cosft2
END INTERFACE
INTERFACE
```

```
    SUBROUTINE covsrt(covar,maska)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: covar
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    END SUBROUTINE covsrt
END INTERFACE
INTERFACE
    SUBROUTINE cyclic(a,b,c,alpha,beta,r,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN):: a,b,c,r
    REAL(SP), INTENT(IN) :: alpha,beta
    REAL(SP), DIMENSION(:), INTENT(OUT):: x
    END SUBROUTINE cyclic
END INTERFACE
INTERFACE
    SUBROUTINE daub4(a,isign)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE daub4
END INTERFACE
INTERFACE dawson
    FUNCTION dawson_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: dawson_s
    END FUNCTION dawson_s

    FUNCTION dawson_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: dawson_v
    END FUNCTION dawson_v
END INTERFACE
INTERFACE
    FUNCTION dbrent(ax,bx,cx,func,dbrent_dfunc,tol,xmin)
    USE nrtype
    REAL(SP), INTENT(IN) :: ax,bx,cx,tol
    REAL(SP), INTENT(OUT) :: xmin
    REAL(SP) :: dbrent
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func

        FUNCTION dbrent_dfunc(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: dbrent_dfunc
        END FUNCTION dbrent_dfunc
    END INTERFACE
    END FUNCTION dbrent
END INTERFACE
INTERFACE
    SUBROUTINE ddpoly(c,x,pd)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(:), INTENT(OUT) :: pd
    END SUBROUTINE ddpoly
END INTERFACE
INTERFACE
    FUNCTION decchk(string,ch)
```

```
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: string
    CHARACTER(1), INTENT(OUT) :: ch
    LOGICAL(LGT) :: decchk
    END FUNCTION decchk
END INTERFACE
INTERFACE
    SUBROUTINE dfpmin(p,gtol,iter,fret,func,dfunc)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: gtol
    REAL(SP), INTENT(OUT) :: fret
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
    INTERFACE
        FUNCTION func(p)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: p
        REAL(SP) :: func
        END FUNCTION func

        FUNCTION dfunc(p)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: p
        REAL(SP), DIMENSION(size(p)) :: dfunc
        END FUNCTION dfunc
    END INTERFACE
    END SUBROUTINE dfpmin
END INTERFACE
INTERFACE
    FUNCTION dfridr(func,x,h,err)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,h
    REAL(SP), INTENT(OUT) :: err
    REAL(SP) :: dfridr
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION dfridr
END INTERFACE
INTERFACE
    SUBROUTINE dftcor(w,delta,a,b,endpts,corre,corim,corfac)
    USE nrtype
    REAL(SP), INTENT(IN) :: w,delta,a,b
    REAL(SP), INTENT(OUT) :: corre,corim,corfac
    REAL(SP), DIMENSION(:), INTENT(IN) :: endpts
    END SUBROUTINE dftcor
END INTERFACE
INTERFACE
    SUBROUTINE dftint(func,a,b,w,cosint,sinint)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,w
    REAL(SP), INTENT(OUT) :: cosint,sinint
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE dftint
```

```
END INTERFACE
INTERFACE
    SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,s,y)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: is1,isf,jsf,k,k1,k2
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: s
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: y
    END SUBROUTINE difeq
END INTERFACE
INTERFACE
    FUNCTION eclass(lista,listb,n)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: lista,listb
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B), DIMENSION(n) :: eclass
    END FUNCTION eclass
END INTERFACE
INTERFACE
    FUNCTION eclazz(equiv,n)
    USE nrtype
    INTERFACE
        FUNCTION equiv(i,j)
        USE nrtype
        LOGICAL(LGT) :: equiv
        INTEGER(I4B), INTENT(IN) :: i,j
        END FUNCTION equiv
    END INTERFACE
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B), DIMENSION(n) :: eclazz
    END FUNCTION eclazz
END INTERFACE
INTERFACE
    FUNCTION ei(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: ei
    END FUNCTION ei
END INTERFACE
INTERFACE
    SUBROUTINE eigsrt(d,v)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: v
    END SUBROUTINE eigsrt
END INTERFACE
INTERFACE elle
    FUNCTION elle_s(phi,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: phi,ak
    REAL(SP) :: elle_s
    END FUNCTION elle_s

    FUNCTION elle_v(phi,ak)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
    REAL(SP), DIMENSION(size(phi)) :: elle_v
    END FUNCTION elle_v
END INTERFACE
INTERFACE ellf
    FUNCTION ellf_s(phi,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: phi,ak
    REAL(SP) :: ellf_s
```

```
    END FUNCTION ellf_s

    FUNCTION ellf_v(phi,ak)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
    REAL(SP), DIMENSION(size(phi)) :: ellf_v
    END FUNCTION ellf_v
END INTERFACE
INTERFACE ellpi
    FUNCTION ellpi_s(phi,en,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: phi,en,ak
    REAL(SP) :: ellpi_s
    END FUNCTION ellpi_s

    FUNCTION ellpi_v(phi,en,ak)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: phi,en,ak
    REAL(SP), DIMENSION(size(phi)) :: ellpi_v
    END FUNCTION ellpi_v
END INTERFACE
INTERFACE
    SUBROUTINE elmhes(a)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    END SUBROUTINE elmhes
END INTERFACE
INTERFACE erf
    FUNCTION erf_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: erf_s
    END FUNCTION erf_s

    FUNCTION erf_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: erf_v
    END FUNCTION erf_v
END INTERFACE
INTERFACE erfc
    FUNCTION erfc_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: erfc_s
    END FUNCTION erfc_s

    FUNCTION erfc_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: erfc_v
    END FUNCTION erfc_v
END INTERFACE
INTERFACE erfcc
    FUNCTION erfcc_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: erfcc_s
    END FUNCTION erfcc_s

    FUNCTION erfcc_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: erfcc_v
    END FUNCTION erfcc_v
END INTERFACE
INTERFACE
```

```
    SUBROUTINE eulsum(sum,term,jterm)
    USE nrtype
    REAL(SP), INTENT(INOUT) :: sum
    REAL(SP), INTENT(IN) :: term
    INTEGER(I4B), INTENT(IN) :: jterm
    END SUBROUTINE eulsum
END INTERFACE
INTERFACE
    FUNCTION evlmem(fdt,d,xms)
    USE nrtype
    REAL(SP), INTENT(IN) :: fdt,xms
    REAL(SP), DIMENSION(:), INTENT(IN) :: d
    REAL(SP) :: evlmem
    END FUNCTION evlmem
END INTERFACE
INTERFACE expdev
    SUBROUTINE expdev_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE expdev_s

    SUBROUTINE expdev_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE expdev_v
END INTERFACE
INTERFACE
    FUNCTION expint(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: expint
    END FUNCTION expint
END INTERFACE
INTERFACE factln
    FUNCTION factln_s(n)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP) :: factln_s
    END FUNCTION factln_s

    FUNCTION factln_v(n)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
    REAL(SP), DIMENSION(size(n)) :: factln_v
    END FUNCTION factln_v
END INTERFACE
INTERFACE factrl
    FUNCTION factrl_s(n)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP) :: factrl_s
    END FUNCTION factrl_s

    FUNCTION factrl_v(n)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
    REAL(SP), DIMENSION(size(n)) :: factrl_v
    END FUNCTION factrl_v
END INTERFACE
INTERFACE
    SUBROUTINE fasper(x,y,ofac,hifac,px,py,jmax,prob)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), INTENT(IN) :: ofac,hifac
    INTEGER(I4B), INTENT(OUT) :: jmax
```

```
    REAL(SP), INTENT(OUT) :: prob
    REAL(SP), DIMENSION(:), POINTER :: px,py
    END SUBROUTINE fasper
END INTERFACE
INTERFACE
    SUBROUTINE fdjac(x,fvec,df)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: fvec
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: df
    END SUBROUTINE fdjac
END INTERFACE
INTERFACE
    SUBROUTINE fgauss(x,a,y,dyda)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: y
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
    END SUBROUTINE fgauss
END INTERFACE
INTERFACE
    SUBROUTINE fit(x,y,a,b,siga,sigb,chi2,q,sig)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
    REAL(SP), DIMENSION(:), OPTIONAL, INTENT(IN) :: sig
    END SUBROUTINE fit
END INTERFACE
INTERFACE
    SUBROUTINE fitexy(x,y,sigx,sigy,a,b,siga,sigb,chi2,q)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sigx,sigy
    REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
    END SUBROUTINE fitexy
END INTERFACE
INTERFACE
    SUBROUTINE fixrts(d)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
    END SUBROUTINE fixrts
END INTERFACE
INTERFACE
    FUNCTION fleg(x,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: fleg
    END FUNCTION fleg
END INTERFACE
INTERFACE
    SUBROUTINE flmoon(n,nph,jd,frac)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n,nph
    INTEGER(I4B), INTENT(OUT) :: jd
    REAL(SP), INTENT(OUT) :: frac
    END SUBROUTINE flmoon
END INTERFACE
INTERFACE four1
    SUBROUTINE four1_dp(data,isign)
    USE nrtype
    COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_dp
```

```
    SUBROUTINE four1_sp(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_sp
END INTERFACE
INTERFACE
    SUBROUTINE four1_alt(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_alt
END INTERFACE
INTERFACE
    SUBROUTINE four1_gather(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_gather
END INTERFACE
INTERFACE
    SUBROUTINE four2(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B),INTENT(IN) :: isign
    END SUBROUTINE four2
END INTERFACE
INTERFACE
    SUBROUTINE four2_alt(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four2_alt
END INTERFACE
INTERFACE
    SUBROUTINE four3(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B),INTENT(IN) :: isign
    END SUBROUTINE four3
END INTERFACE
INTERFACE
    SUBROUTINE four3_alt(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four3_alt
END INTERFACE
INTERFACE
    SUBROUTINE fourcol(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourcol
END INTERFACE
INTERFACE
    SUBROUTINE fourcol_3d(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourcol_3d
END INTERFACE
INTERFACE
    SUBROUTINE fourn_gather(data,nn,isign)
```

```
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourn_gather
END INTERFACE
INTERFACE fourrow
    SUBROUTINE fourrow_dp(data,isign)
    USE nrtype
    COMPLEX(DPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourrow_dp

    SUBROUTINE fourrow_sp(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourrow_sp
END INTERFACE
INTERFACE
    SUBROUTINE fourrow_3d(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourrow_3d
END INTERFACE
INTERFACE
    FUNCTION fpoly(x,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: fpoly
    END FUNCTION fpoly
END INTERFACE
INTERFACE
    SUBROUTINE fred2(a,b,t,f,w,g,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(OUT) :: t,f,w
    INTERFACE
        FUNCTION g(t)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: t
        REAL(SP), DIMENSION(size(t)) :: g
        END FUNCTION g

        FUNCTION ak(t,s)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
        REAL(SP), DIMENSION(size(t),size(s)) :: ak
        END FUNCTION ak
    END INTERFACE
    END SUBROUTINE fred2
END INTERFACE
INTERFACE
    FUNCTION fredin(x,a,b,t,f,w,g,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,t,f,w
    REAL(SP), DIMENSION(size(x)) :: fredin
    INTERFACE
        FUNCTION g(t)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: t
        REAL(SP), DIMENSION(size(t)) :: g
```

```
        END FUNCTION g

        FUNCTION ak(t,s)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
        REAL(SP), DIMENSION(size(t),size(s)) :: ak
        END FUNCTION ak
    END INTERFACE
    END FUNCTION fredin
END INTERFACE
INTERFACE
    SUBROUTINE frenel(x,s,c)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: s,c
    END SUBROUTINE frenel
END INTERFACE
INTERFACE
    SUBROUTINE frprmn(p,ftol,iter,fret)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: ftol
    REAL(SP), INTENT(OUT) :: fret
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
    END SUBROUTINE frprmn
END INTERFACE
INTERFACE
    SUBROUTINE ftest(data1,data2,f,prob)
    USE nrtype
    REAL(SP), INTENT(OUT) :: f,prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    END SUBROUTINE ftest
END INTERFACE
INTERFACE
    FUNCTION gamdev(ia)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: ia
    REAL(SP) :: gamdev
    END FUNCTION gamdev
END INTERFACE
INTERFACE gammln
    FUNCTION gammln_s(xx)
    USE nrtype
    REAL(SP), INTENT(IN) :: xx
    REAL(SP) :: gammln_s
    END FUNCTION gammln_s

    FUNCTION gammln_v(xx)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xx
    REAL(SP), DIMENSION(size(xx)) :: gammln_v
    END FUNCTION gammln_v
END INTERFACE
INTERFACE gammp
    FUNCTION gammp_s(a,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP) :: gammp_s
    END FUNCTION gammp_s

    FUNCTION gammp_v(a,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(size(a)) :: gammp_v
    END FUNCTION gammp_v
END INTERFACE
```

```
INTERFACE gammq
    FUNCTION gammq_s(a,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP) :: gammq_s
    END FUNCTION gammq_s

    FUNCTION gammq_v(a,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(size(a)) :: gammq_v
    END FUNCTION gammq_v
END INTERFACE
INTERFACE gasdev
    SUBROUTINE gasdev_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE gasdev_s

    SUBROUTINE gasdev_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE gasdev_v
END INTERFACE
INTERFACE
    SUBROUTINE gaucof(a,b,amu0,x,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: amu0
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaucof
END INTERFACE
INTERFACE
    SUBROUTINE gauher(x,w)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gauher
END INTERFACE
INTERFACE
    SUBROUTINE gaujac(x,w,alf,bet)
    USE nrtype
    REAL(SP), INTENT(IN) :: alf,bet
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaujac
END INTERFACE
INTERFACE
    SUBROUTINE gaulag(x,w,alf)
    USE nrtype
    REAL(SP), INTENT(IN) :: alf
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaulag
END INTERFACE
INTERFACE
    SUBROUTINE gauleg(x1,x2,x,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gauleg
END INTERFACE
INTERFACE
    SUBROUTINE gaussj(a,b)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
    END SUBROUTINE gaussj
END INTERFACE
```

```
INTERFACE gcf
    FUNCTION gcf_s(a,x,gln)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP) :: gcf_s
    END FUNCTION gcf_s
    FUNCTION gcf_v(a,x,gln)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP), DIMENSION(size(a)) :: gcf_v
    END FUNCTION gcf_v
END INTERFACE
INTERFACE
    FUNCTION golden(ax,bx,cx,func,tol,xmin)
    USE nrtype
    REAL(SP), INTENT(IN) :: ax,bx,cx,tol
    REAL(SP), INTENT(OUT) :: xmin
    REAL(SP) :: golden
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION golden
END INTERFACE
INTERFACE gser
    FUNCTION gser_s(a,x,gln)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP) :: gser_s
    END FUNCTION gser_s
    FUNCTION gser_v(a,x,gln)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP), DIMENSION(size(a)) :: gser_v
    END FUNCTION gser_v
END INTERFACE
INTERFACE
    SUBROUTINE hqr(a,wr,wi)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: wr,wi
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    END SUBROUTINE hqr
END INTERFACE
INTERFACE
    SUBROUTINE hunt(xx,x,jlo)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: jlo
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: xx
    END SUBROUTINE hunt
END INTERFACE
INTERFACE
    SUBROUTINE hypdrv(s,ry,rdyds)
    USE nrtype
    REAL(SP), INTENT(IN) :: s
    REAL(SP), DIMENSION(:), INTENT(IN) :: ry
```

```
    REAL(SP), DIMENSION(:), INTENT(OUT) :: rdyds
    END SUBROUTINE hypdrv
END INTERFACE
INTERFACE
    FUNCTION hypgeo(a,b,c,z)
    USE nrtype
    COMPLEX(SPC), INTENT(IN) :: a,b,c,z
    COMPLEX(SPC) :: hypgeo
    END FUNCTION hypgeo
END INTERFACE
INTERFACE
    SUBROUTINE hypser(a,b,c,z,series,deriv)
    USE nrtype
    COMPLEX(SPC), INTENT(IN) :: a,b,c,z
    COMPLEX(SPC), INTENT(OUT) :: series,deriv
    END SUBROUTINE hypser
END INTERFACE
INTERFACE
    FUNCTION icrc(crc,buf,jinit,jrev)
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: buf
    INTEGER(I2B), INTENT(IN) :: crc,jinit
    INTEGER(I4B), INTENT(IN) :: jrev
    INTEGER(I2B) :: icrc
    END FUNCTION icrc
END INTERFACE
INTERFACE
    FUNCTION igray(n,is)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n,is
    INTEGER(I4B) :: igray
    END FUNCTION igray
END INTERFACE
INTERFACE
    RECURSIVE SUBROUTINE index_bypack(arr,index,partial)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: arr
    INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: index
    INTEGER, OPTIONAL, INTENT(IN) :: partial
    END SUBROUTINE index_bypack
END INTERFACE
INTERFACE indexx
    SUBROUTINE indexx_sp(arr,index)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: arr
    INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
    END SUBROUTINE indexx_sp
    SUBROUTINE indexx_i4b(iarr,index)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
    INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
    END SUBROUTINE indexx_i4b
END INTERFACE
INTERFACE
    FUNCTION interp(uc)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: uc
    REAL(DP), DIMENSION(2*size(uc,1)-1,2*size(uc,1)-1) :: interp
    END FUNCTION interp
END INTERFACE
INTERFACE
    FUNCTION rank(indx)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
```

```
    INTEGER(I4B), DIMENSION(size(indx)) :: rank
    END FUNCTION rank
END INTERFACE
INTERFACE
    FUNCTION irbit1(iseed)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: iseed
    INTEGER(I4B) :: irbit1
    END FUNCTION irbit1
END INTERFACE
INTERFACE
    FUNCTION irbit2(iseed)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: iseed
    INTEGER(I4B) :: irbit2
    END FUNCTION irbit2
END INTERFACE
INTERFACE
    SUBROUTINE jacobi(a,d,v,nrot)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: nrot
    REAL(SP), DIMENSION(:), INTENT(OUT) :: d
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
    END SUBROUTINE jacobi
END INTERFACE
INTERFACE
    SUBROUTINE jacobn(x,y,dfdx,dfdy)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
    END SUBROUTINE jacobn
END INTERFACE
INTERFACE
    FUNCTION julday(mm,id,iyyy)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: mm,id,iyyy
    INTEGER(I4B) :: julday
    END FUNCTION julday
END INTERFACE
INTERFACE
    SUBROUTINE kendl1(data1,data2,tau,z,prob)
    USE nrtype
    REAL(SP), INTENT(OUT) :: tau,z,prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    END SUBROUTINE kendl1
END INTERFACE
INTERFACE
    SUBROUTINE kendl2(tab,tau,z,prob)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: tab
    REAL(SP), INTENT(OUT) :: tau,z,prob
    END SUBROUTINE kendl2
END INTERFACE
INTERFACE
    FUNCTION kermom(y,m)
    USE nrtype
    REAL(DP), INTENT(IN) :: y
    INTEGER(I4B), INTENT(IN) :: m
    REAL(DP), DIMENSION(m) :: kermom
    END FUNCTION kermom
END INTERFACE
```

```
INTERFACE
    SUBROUTINE ks2d1s(x1,y1,quadvl,d1,prob)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1
    REAL(SP), INTENT(OUT) :: d1,prob
    INTERFACE
        SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
        USE nrtype
        REAL(SP), INTENT(IN) :: x,y
        REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
        END SUBROUTINE quadvl
    END INTERFACE
    END SUBROUTINE ks2d1s
END INTERFACE
INTERFACE
    SUBROUTINE ks2d2s(x1,y1,x2,y2,d,prob)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1,x2,y2
    REAL(SP), INTENT(OUT) :: d,prob
    END SUBROUTINE ks2d2s
END INTERFACE
INTERFACE
    SUBROUTINE ksone(data,func,d,prob)
    USE nrtype
    REAL(SP), INTENT(OUT) :: d,prob
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE ksone
END INTERFACE
INTERFACE
    SUBROUTINE kstwo(data1,data2,d,prob)
    USE nrtype
    REAL(SP), INTENT(OUT) :: d,prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    END SUBROUTINE kstwo
END INTERFACE
INTERFACE
    SUBROUTINE laguer(a,x,its)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: its
    COMPLEX(SPC), INTENT(INOUT) :: x
    COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
    END SUBROUTINE laguer
END INTERFACE
INTERFACE
    SUBROUTINE lfit(x,y,sig,a,maska,covar,chisq,funcs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: covar
    REAL(SP), INTENT(OUT) :: chisq
    INTERFACE
        SUBROUTINE funcs(x,arr)
        USE nrtype
        REAL(SP),INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(OUT) :: arr
        END SUBROUTINE funcs
```

```
    END INTERFACE
    END SUBROUTINE lfit
END INTERFACE
INTERFACE
    SUBROUTINE linbcg(b,x,itol,tol,itmax,iter,err)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: b
    REAL(DP), DIMENSION(:), INTENT(INOUT) :: x
    INTEGER(I4B), INTENT(IN) :: itol,itmax
    REAL(DP), INTENT(IN) :: tol
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(DP), INTENT(OUT) :: err
    END SUBROUTINE linbcg
END INTERFACE
INTERFACE
    SUBROUTINE linmin(p,xi,fret)
    USE nrtype
    REAL(SP), INTENT(OUT) :: fret
    REAL(SP), DIMENSION(:), TARGET, INTENT(INOUT) :: p,xi
    END SUBROUTINE linmin
END INTERFACE
INTERFACE
    SUBROUTINE lnsrch(xold,fold,g,p,x,f,stpmax,check,func)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xold,g
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
    REAL(SP), INTENT(IN) :: fold,stpmax
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x
    REAL(SP), INTENT(OUT) :: f
    LOGICAL(LGT), INTENT(OUT) :: check
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP) :: func
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE lnsrch
END INTERFACE
INTERFACE
    FUNCTION locate(xx,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xx
    REAL(SP), INTENT(IN) :: x
    INTEGER(I4B) :: locate
    END FUNCTION locate
END INTERFACE
INTERFACE
    FUNCTION lop(u)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: u
    REAL(DP), DIMENSION(size(u,1),size(u,1)) :: lop
    END FUNCTION lop
END INTERFACE
INTERFACE
    SUBROUTINE lubksb(a,indx,b)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
    END SUBROUTINE lubksb
END INTERFACE
INTERFACE
    SUBROUTINE ludcmp(a,indx,d)
```

```
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
    REAL(SP), INTENT(OUT) :: d
    END SUBROUTINE ludcmp
END INTERFACE
INTERFACE
    SUBROUTINE machar(ibeta,it,irnd,ngrd,machep,negep,iexp,minexp,&
        maxexp,eps,epsneg,xmin,xmax)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: ibeta,iexp,irnd,it,machep,maxexp,&
        minexp,negep,ngrd
    REAL(SP), INTENT(OUT) :: eps,epsneg,xmax,xmin
    END SUBROUTINE machar
END INTERFACE
INTERFACE
    SUBROUTINE medfit(x,y,a,b,abdev)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: a,b,abdev
    END SUBROUTINE medfit
END INTERFACE
INTERFACE
    SUBROUTINE memcof(data,xms,d)
    USE nrtype
    REAL(SP), INTENT(OUT) :: xms
    REAL(SP), DIMENSION(:), INTENT(IN) :: data
    REAL(SP), DIMENSION(:), INTENT(OUT) :: d
    END SUBROUTINE memcof
END INTERFACE
INTERFACE
    SUBROUTINE mgfas(u,maxcyc)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
    INTEGER(I4B), INTENT(IN) :: maxcyc
    END SUBROUTINE mgfas
END INTERFACE
INTERFACE
    SUBROUTINE mglin(u,ncycle)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
    INTEGER(I4B), INTENT(IN) :: ncycle
    END SUBROUTINE mglin
END INTERFACE
INTERFACE
    SUBROUTINE midexp(funk,aa,bb,s,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: aa,bb
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
        FUNCTION funk(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: funk
        END FUNCTION funk
    END INTERFACE
    END SUBROUTINE midexp
END INTERFACE
INTERFACE
    SUBROUTINE midinf(funk,aa,bb,s,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: aa,bb
    REAL(SP), INTENT(INOUT) :: s
```

```
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
        FUNCTION funk(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: funk
        END FUNCTION funk
    END INTERFACE
    END SUBROUTINE midinf
END INTERFACE
INTERFACE
    SUBROUTINE midpnt(func,a,b,s,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE midpnt
END INTERFACE
INTERFACE
    SUBROUTINE midsql(funk,aa,bb,s,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: aa,bb
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
        FUNCTION funk(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: funk
        END FUNCTION funk
    END INTERFACE
    END SUBROUTINE midsql
END INTERFACE
INTERFACE
    SUBROUTINE midsqu(funk,aa,bb,s,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: aa,bb
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
        FUNCTION funk(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: funk
        END FUNCTION funk
    END INTERFACE
    END SUBROUTINE midsqu
END INTERFACE
INTERFACE
    RECURSIVE SUBROUTINE miser(func,regn,ndim,npts,dith,ave,var)
    USE nrtype
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP) :: func
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        END FUNCTION func
```

```
    END INTERFACE
    REAL(SP), DIMENSION(:), INTENT(IN) :: regn
    INTEGER(I4B), INTENT(IN) :: ndim,npts
    REAL(SP), INTENT(IN) :: dith
    REAL(SP), INTENT(OUT) :: ave,var
    END SUBROUTINE miser
END INTERFACE
INTERFACE
    SUBROUTINE mmid(y,dydx,xs,htot,nstep,yout,derivs)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: nstep
    REAL(SP), INTENT(IN) :: xs,htot
    REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE mmid
END INTERFACE
INTERFACE
    SUBROUTINE mnbrak(ax,bx,cx,fa,fb,fc,func)
    USE nrtype
    REAL(SP), INTENT(INOUT) :: ax,bx
    REAL(SP), INTENT(OUT) :: cx,fa,fb,fc
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE mnbrak
END INTERFACE
INTERFACE
    SUBROUTINE mnewt(ntrial,x,tolx,tolf,usrfun)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: ntrial
    REAL(SP), INTENT(IN) :: tolx,tolf
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    INTERFACE
        SUBROUTINE usrfun(x,fvec,fjac)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(OUT) :: fvec
        REAL(SP), DIMENSION(:,:), INTENT(OUT) :: fjac
        END SUBROUTINE usrfun
    END INTERFACE
    END SUBROUTINE mnewt
END INTERFACE
INTERFACE
    SUBROUTINE moment(data,ave,adev,sdev,var,skew,curt)
    USE nrtype
    REAL(SP), INTENT(OUT) :: ave,adev,sdev,var,skew,curt
    REAL(SP), DIMENSION(:), INTENT(IN) :: data
    END SUBROUTINE moment
END INTERFACE
INTERFACE
    SUBROUTINE mp2dfr(a,s,n,m)
    USE nrtype
```

```
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B), INTENT(OUT) :: m
    CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: a
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: s
    END SUBROUTINE mp2dfr
END INTERFACE
INTERFACE
    SUBROUTINE mpdiv(q,r,u,v,n,m)
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: q,r
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
    INTEGER(I4B), INTENT(IN) :: n,m
    END SUBROUTINE mpdiv
END INTERFACE
INTERFACE
    SUBROUTINE mpinv(u,v,n,m)
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: u
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
    INTEGER(I4B), INTENT(IN) :: n,m
    END SUBROUTINE mpinv
END INTERFACE
INTERFACE
    SUBROUTINE mpmul(w,u,v,n,m)
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
    INTEGER(I4B), INTENT(IN) :: n,m
    END SUBROUTINE mpmul
END INTERFACE
INTERFACE
    SUBROUTINE mppi(n)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    END SUBROUTINE mppi
END INTERFACE
INTERFACE
    SUBROUTINE mprove(a,alud,indx,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,alud
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
    REAL(SP), DIMENSION(:), INTENT(IN) :: b
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    END SUBROUTINE mprove
END INTERFACE
INTERFACE
    SUBROUTINE mpsqrt(w,u,v,n,m)
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w,u
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
    INTEGER(I4B), INTENT(IN) :: n,m
    END SUBROUTINE mpsqrt
END INTERFACE
INTERFACE
    SUBROUTINE mrqcof(x,y,sig,a,maska,alpha,beta,chisq,funcs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,a,sig
    REAL(SP), DIMENSION(:), INTENT(OUT) :: beta
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: alpha
    REAL(SP), INTENT(OUT) :: chisq
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    INTERFACE
        SUBROUTINE funcs(x,a,yfit,dyda)
        USE nrtype
```

```
        REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
        REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
        REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
        END SUBROUTINE funcs
    END INTERFACE
    END SUBROUTINE mrqcof
END INTERFACE
INTERFACE
    SUBROUTINE mrqmin(x,y,sig,a,maska,covar,alpha,chisq,funcs,alamda)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: covar,alpha
    REAL(SP), INTENT(OUT) :: chisq
    REAL(SP), INTENT(INOUT) :: alamda
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    INTERFACE
        SUBROUTINE funcs(x,a,yfit,dyda)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
        REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
        REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
        END SUBROUTINE funcs
    END INTERFACE
    END SUBROUTINE mrqmin
END INTERFACE
INTERFACE
    SUBROUTINE newt(x,check)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    LOGICAL(LGT), INTENT(OUT) :: check
    END SUBROUTINE newt
END INTERFACE
INTERFACE
    SUBROUTINE odeint(ystart,x1,x2,eps,h1,hmin,derivs,rkqs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: ystart
    REAL(SP), INTENT(IN) :: x1,x2,eps,h1,hmin
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs

        SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
        REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
        REAL(SP), INTENT(INOUT) :: x
        REAL(SP), INTENT(IN) :: htry,eps
        REAL(SP), INTENT(OUT) :: hdid,hnext
            INTERFACE
            SUBROUTINE derivs(x,y,dydx)
                USE nrtype
                REAL(SP), INTENT(IN) :: x
                REAL(SP), DIMENSION(:), INTENT(IN) :: y
                REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
                END SUBROUTINE derivs
            END INTERFACE
        END SUBROUTINE rkqs
    END INTERFACE
    END SUBROUTINE odeint
```

```
END INTERFACE
INTERFACE
    SUBROUTINE orthog(anu,alpha,beta,a,b)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: anu,alpha,beta
    REAL(SP), DIMENSION(:), INTENT(OUT) :: a,b
    END SUBROUTINE orthog
END INTERFACE
INTERFACE
    SUBROUTINE pade(cof,resid)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(INOUT) :: cof
    REAL(SP), INTENT(OUT) :: resid
    END SUBROUTINE pade
END INTERFACE
INTERFACE
    FUNCTION pccheb(d)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: d
    REAL(SP), DIMENSION(size(d)) :: pccheb
    END FUNCTION pccheb
END INTERFACE
INTERFACE
    SUBROUTINE pcshft(a,b,d)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
    END SUBROUTINE pcshft
END INTERFACE
INTERFACE
    SUBROUTINE pearsn(x,y,r,prob,z)
    USE nrtype
    REAL(SP), INTENT(OUT) :: r,prob,z
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    END SUBROUTINE pearsn
END INTERFACE
INTERFACE
    SUBROUTINE period(x,y,ofac,hifac,px,py,jmax,prob)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: jmax
    REAL(SP), INTENT(IN) :: ofac,hifac
    REAL(SP), INTENT(OUT) :: prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), POINTER :: px,py
    END SUBROUTINE period
END INTERFACE
INTERFACE plgndr
    FUNCTION plgndr_s(l,m,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: l,m
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: plgndr_s
    END FUNCTION plgndr_s
    FUNCTION plgndr_v(l,m,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: l,m
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: plgndr_v
    END FUNCTION plgndr_v
END INTERFACE
INTERFACE
    FUNCTION poidev(xm)
    USE nrtype
```

```
    REAL(SP), INTENT(IN) :: xm
    REAL(SP) :: poidev
    END FUNCTION poidev
END INTERFACE
INTERFACE
    FUNCTION polcoe(x,y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(size(x)) :: polcoe
    END FUNCTION polcoe
END INTERFACE
INTERFACE
    FUNCTION polcof(xa,ya)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
    REAL(SP), DIMENSION(size(xa)) :: polcof
    END FUNCTION polcof
END INTERFACE
INTERFACE
    SUBROUTINE poldiv(u,v,q,r)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: u,v
    REAL(SP), DIMENSION(:), INTENT(OUT) :: q,r
    END SUBROUTINE poldiv
END INTERFACE
INTERFACE
    SUBROUTINE polin2(x1a,x2a,ya,x1,x2,y,dy)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP), INTENT(OUT) :: y,dy
    END SUBROUTINE polin2
END INTERFACE
INTERFACE
    SUBROUTINE polint(xa,ya,x,y,dy)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: y,dy
    END SUBROUTINE polint
END INTERFACE
INTERFACE
    SUBROUTINE powell(p,xi,ftol,iter,fret)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: xi
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: ftol
    REAL(SP), INTENT(OUT) :: fret
    END SUBROUTINE powell
END INTERFACE
INTERFACE
    FUNCTION predic(data,d,nfut)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data,d
    INTEGER(I4B), INTENT(IN) :: nfut
    REAL(SP), DIMENSION(nfut) :: predic
    END FUNCTION predic
END INTERFACE
INTERFACE
    FUNCTION probks(alam)
    USE nrtype
    REAL(SP), INTENT(IN) :: alam
```

```
    REAL(SP) :: probks
    END FUNCTION probks
END INTERFACE
INTERFACE psdes
    SUBROUTINE psdes_s(lword,rword)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: lword,rword
    END SUBROUTINE psdes_s

    SUBROUTINE psdes_v(lword,rword)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: lword,rword
    END SUBROUTINE psdes_v
END INTERFACE
INTERFACE
    SUBROUTINE pwt(a,isign)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE pwt
END INTERFACE
INTERFACE
    SUBROUTINE pwtset(n)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    END SUBROUTINE pwtset
END INTERFACE
INTERFACE pythag
    FUNCTION pythag_dp(a,b)
    USE nrtype
    REAL(DP), INTENT(IN) :: a,b
    REAL(DP) :: pythag_dp
    END FUNCTION pythag_dp

    FUNCTION pythag_sp(a,b)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP) :: pythag_sp
    END FUNCTION pythag_sp
END INTERFACE
INTERFACE
    SUBROUTINE pzextr(iest,xest,yest,yz,dy)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: iest
    REAL(SP), INTENT(IN) :: xest
    REAL(SP), DIMENSION(:), INTENT(IN) :: yest
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
    END SUBROUTINE pzextr
END INTERFACE
INTERFACE
    SUBROUTINE qrdcmp(a,c,d,sing)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: c,d
    LOGICAL(LGT), INTENT(OUT) :: sing
    END SUBROUTINE qrdcmp
END INTERFACE
INTERFACE
    FUNCTION qromb(func,a,b)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP) :: qromb
    INTERFACE
        FUNCTION func(x)
        USE nrtype
```

```
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION qromb
END INTERFACE
INTERFACE
    FUNCTION qromo(func,a,b,choose)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP) :: qromo
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    INTERFACE
        SUBROUTINE choose(funk,aa,bb,s,n)
        USE nrtype
        REAL(SP), INTENT(IN) :: aa,bb
        REAL(SP), INTENT(INOUT) :: s
        INTEGER(I4B), INTENT(IN) :: n
        INTERFACE
            FUNCTION funk(x)
            USE nrtype
            REAL(SP), DIMENSION(:), INTENT(IN) :: x
            REAL(SP), DIMENSION(size(x)) :: funk
            END FUNCTION funk
        END INTERFACE
        END SUBROUTINE choose
    END INTERFACE
    END FUNCTION qromo
END INTERFACE
INTERFACE
    SUBROUTINE qroot(p,b,c,eps)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP), INTENT(INOUT) :: b,c
    REAL(SP), INTENT(IN) :: eps
    END SUBROUTINE qroot
END INTERFACE
INTERFACE
    SUBROUTINE qrsolv(a,c,d,b)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
    REAL(SP), DIMENSION(:), INTENT(IN) :: c,d
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
    END SUBROUTINE qrsolv
END INTERFACE
INTERFACE
    SUBROUTINE qrupdt(r,qt,u,v)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: r,qt
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: u
    REAL(SP), DIMENSION(:), INTENT(IN) :: v
    END SUBROUTINE qrupdt
END INTERFACE
INTERFACE
    FUNCTION qsimp(func,a,b)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP) :: qsimp
```

```
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION qsimp
END INTERFACE
INTERFACE
    FUNCTION qtrap(func,a,b)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP) :: qtrap
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION qtrap
END INTERFACE
INTERFACE
    SUBROUTINE quadct(x,y,xx,yy,fa,fb,fc,fd)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), INTENT(IN) :: xx,yy
    REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    END SUBROUTINE quadct
END INTERFACE
INTERFACE
    SUBROUTINE quadmx(a)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: a
    END SUBROUTINE quadmx
END INTERFACE
INTERFACE
    SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    END SUBROUTINE quadvl
END INTERFACE
INTERFACE
    FUNCTION ran(idum)
    INTEGER(selected_int_kind(9)), INTENT(INOUT) :: idum
    REAL :: ran
    END FUNCTION ran
END INTERFACE
INTERFACE ran0
    SUBROUTINE ran0_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran0_s
    SUBROUTINE ran0_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran0_v
END INTERFACE
INTERFACE ran1
    SUBROUTINE ran1_s(harvest)
    USE nrtype
```

```
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran1_s
    SUBROUTINE ran1_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran1_v
END INTERFACE
INTERFACE ran2
    SUBROUTINE ran2_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran2_s
    SUBROUTINE ran2_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran2_v
END INTERFACE
INTERFACE ran3
    SUBROUTINE ran3_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran3_s
    SUBROUTINE ran3_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran3_v
END INTERFACE
INTERFACE
    SUBROUTINE ratint(xa,ya,x,y,dy)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: y,dy
    END SUBROUTINE ratint
END INTERFACE
INTERFACE
    SUBROUTINE ratlsq(func,a,b,mm,kk,cof,dev)
    USE nrtype
    REAL(DP), INTENT(IN) :: a,b
    INTEGER(I4B), INTENT(IN) :: mm,kk
    REAL(DP), DIMENSION(:), INTENT(OUT) :: cof
    REAL(DP), INTENT(OUT) :: dev
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(DP), DIMENSION(:), INTENT(IN) :: x
        REAL(DP), DIMENSION(size(x)) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE ratlsq
END INTERFACE
INTERFACE ratval
    FUNCTION ratval_s(x,cof,mm,kk)
    USE nrtype
    REAL(DP), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: mm,kk
    REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
    REAL(DP) :: ratval_s
    END FUNCTION ratval_s
    FUNCTION ratval_v(x,cof,mm,kk)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x
```

```
    INTEGER(I4B), INTENT(IN) :: mm,kk
    REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
    REAL(DP), DIMENSION(size(x)) :: ratval_v
    END FUNCTION ratval_v
END INTERFACE
INTERFACE rc
    FUNCTION rc_s(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: rc_s
    END FUNCTION rc_s

    FUNCTION rc_v(x,y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(size(x)) :: rc_v
    END FUNCTION rc_v
END INTERFACE
INTERFACE rd
    FUNCTION rd_s(x,y,z)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y,z
    REAL(SP) :: rd_s
    END FUNCTION rd_s

    FUNCTION rd_v(x,y,z)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
    REAL(SP), DIMENSION(size(x)) :: rd_v
    END FUNCTION rd_v
END INTERFACE
INTERFACE realft
    SUBROUTINE realft_dp(data,isign,zdata)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    COMPLEX(DPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
    END SUBROUTINE realft_dp

    SUBROUTINE realft_sp(data,isign,zdata)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    COMPLEX(SPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
    END SUBROUTINE realft_sp
END INTERFACE
INTERFACE
    RECURSIVE FUNCTION recur1(a,b) RESULT(u)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(size(a)) :: u
    END FUNCTION recur1
END INTERFACE
INTERFACE
    FUNCTION recur2(a,b,c)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c
    REAL(SP), DIMENSION(size(a)) :: recur2
    END FUNCTION recur2
END INTERFACE
INTERFACE
    SUBROUTINE relax(u,rhs)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
    END SUBROUTINE relax
```

```
END INTERFACE
INTERFACE
    SUBROUTINE relax2(u,rhs)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
    END SUBROUTINE relax2
END INTERFACE
INTERFACE
FUNCTION resid(u,rhs)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,rhs
    REAL(DP), DIMENSION(size(u,1),size(u,1)) :: resid
    END FUNCTION resid
END INTERFACE
INTERFACE rf
    FUNCTION rf_s(x,y,z)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y,z
    REAL(SP) :: rf_s
    END FUNCTION rf_s

    FUNCTION rf_v(x,y,z)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
    REAL(SP), DIMENSION(size(x)) :: rf_v
    END FUNCTION rf_v
END INTERFACE
INTERFACE rj
    FUNCTION rj_s(x,y,z,p)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y,z,p
    REAL(SP) :: rj_s
    END FUNCTION rj_s

    FUNCTION rj_v(x,y,z,p)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z,p
    REAL(SP), DIMENSION(size(x)) :: rj_v
    END FUNCTION rj_v
END INTERFACE
INTERFACE
    SUBROUTINE rk4(y,dydx,x,h,yout,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
    REAL(SP), INTENT(IN) :: x,h
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE rk4
END INTERFACE
INTERFACE
    SUBROUTINE rkck(y,dydx,x,h,yout,yerr,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
    REAL(SP), INTENT(IN) :: x,h
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yout,yerr
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
```

```
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE rkck
END INTERFACE
INTERFACE
    SUBROUTINE rkdumb(vstart,x1,x2,nstep,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: vstart
    REAL(SP), INTENT(IN) :: x1,x2
    INTEGER(I4B), INTENT(IN) :: nstep
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE rkdumb
END INTERFACE
INTERFACE
    SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(OUT) :: hdid,hnext
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE rkqs
END INTERFACE
INTERFACE
    SUBROUTINE rlft2(data,spec,speq,isign)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: data
    COMPLEX(SPC), DIMENSION(:,:), INTENT(OUT) :: spec
    COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: speq
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE rlft2
END INTERFACE
INTERFACE
    SUBROUTINE rlft3(data,spec,speq,isign)
    USE nrtype
    REAL(SP), DIMENSION(:,:,:), INTENT(INOUT) :: data
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(OUT) :: spec
    COMPLEX(SPC), DIMENSION(:,:), INTENT(OUT) :: speq
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE rlft3
END INTERFACE
INTERFACE
    SUBROUTINE rotate(r,qt,i,a,b)
    USE nrtype
    REAL(SP), DIMENSION(:,:), TARGET, INTENT(INOUT) :: r,qt
```

```
    INTEGER(I4B), INTENT(IN) :: i
    REAL(SP), INTENT(IN) :: a,b
    END SUBROUTINE rotate
END INTERFACE
INTERFACE
    SUBROUTINE rsolv(a,d,b)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
    REAL(SP), DIMENSION(:), INTENT(IN) :: d
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
    END SUBROUTINE rsolv
END INTERFACE
INTERFACE
    FUNCTION rstrct(uf)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: uf
    REAL(DP), DIMENSION((size(uf,1)+1)/2,(size(uf,1)+1)/2) :: rstrct
    END FUNCTION rstrct
END INTERFACE
INTERFACE
    FUNCTION rtbis(func,x1,x2,xacc)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: rtbis
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION rtbis
END INTERFACE
INTERFACE
    FUNCTION rtflsp(func,x1,x2,xacc)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: rtflsp
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION rtflsp
END INTERFACE
INTERFACE
    FUNCTION rtnewt(funcd,x1,x2,xacc)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: rtnewt
    INTERFACE
        SUBROUTINE funcd(x,fval,fderiv)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), INTENT(OUT) :: fval,fderiv
        END SUBROUTINE funcd
    END INTERFACE
    END FUNCTION rtnewt
END INTERFACE
INTERFACE
    FUNCTION rtsafe(funcd,x1,x2,xacc)
    USE nrtype
```

```
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: rtsafe
    INTERFACE
        SUBROUTINE funcd(x,fval,fderiv)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), INTENT(OUT) :: fval,fderiv
        END SUBROUTINE funcd
    END INTERFACE
    END FUNCTION rtsafe
END INTERFACE
INTERFACE
    FUNCTION rtsec(func,x1,x2,xacc)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: rtsec
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION rtsec
END INTERFACE
INTERFACE
    SUBROUTINE rzextr(iest,xest,yest,yz,dy)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: iest
    REAL(SP), INTENT(IN) :: xest
    REAL(SP), DIMENSION(:), INTENT(IN) :: yest
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
    END SUBROUTINE rzextr
END INTERFACE
INTERFACE
    FUNCTION savgol(nl,nrr,ld,m)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
    REAL(SP), DIMENSION(nl+nrr+1) :: savgol
    END FUNCTION savgol
END INTERFACE
INTERFACE
    SUBROUTINE scrsho(func)
    USE nrtype
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE scrsho
END INTERFACE
INTERFACE
    FUNCTION select(k,arr)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: k
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    REAL(SP) :: select
    END FUNCTION select
END INTERFACE
INTERFACE
    FUNCTION select_bypack(k,arr)
    USE nrtype
```

```
    INTEGER(I4B), INTENT(IN) :: k
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    REAL(SP) :: select_bypack
    END FUNCTION select_bypack
END INTERFACE
INTERFACE
    SUBROUTINE select_heap(arr,heap)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: arr
    REAL(SP), DIMENSION(:), INTENT(OUT) :: heap
    END SUBROUTINE select_heap
END INTERFACE
INTERFACE
    FUNCTION select_inplace(k,arr)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: k
    REAL(SP), DIMENSION(:), INTENT(IN) :: arr
    REAL(SP) :: select_inplace
    END FUNCTION select_inplace
END INTERFACE
INTERFACE
    SUBROUTINE simplx(a,m1,m2,m3,icase,izrov,iposv)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: m1,m2,m3
    INTEGER(I4B), INTENT(OUT) :: icase
    INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: izrov,iposv
    END SUBROUTINE simplx
END INTERFACE
INTERFACE
    SUBROUTINE simpr(y,dydx,dfdx,dfdy,xs,htot,nstep,yout,derivs)
    USE nrtype
    REAL(SP), INTENT(IN) :: xs,htot
    REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx,dfdx
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: dfdy
    INTEGER(I4B), INTENT(IN) :: nstep
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE simpr
END INTERFACE
INTERFACE
    SUBROUTINE sinft(y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    END SUBROUTINE sinft
END INTERFACE
INTERFACE
    SUBROUTINE slvsm2(u,rhs)
    USE nrtype
    REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
    REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
    END SUBROUTINE slvsm2
END INTERFACE
INTERFACE
    SUBROUTINE slvsml(u,rhs)
    USE nrtype
    REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
```

```
    REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
    END SUBROUTINE slvsml
END INTERFACE
INTERFACE
    SUBROUTINE sncndn(uu,emmc,sn,cn,dn)
    USE nrtype
    REAL(SP), INTENT(IN) :: uu,emmc
    REAL(SP), INTENT(OUT) :: sn,cn,dn
    END SUBROUTINE sncndn
END INTERFACE
INTERFACE
    FUNCTION snrm(sx,itol)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: sx
    INTEGER(I4B), INTENT(IN) :: itol
    REAL(DP) :: snrm
    END FUNCTION snrm
END INTERFACE
INTERFACE
    SUBROUTINE sobseq(x,init)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x
    INTEGER(I4B), OPTIONAL, INTENT(IN) :: init
    END SUBROUTINE sobseq
END INTERFACE
INTERFACE
    SUBROUTINE solvde(itmax,conv,slowc,scalv,indexv,nb,y)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: itmax,nb
    REAL(SP), INTENT(IN) :: conv,slowc
    REAL(SP), DIMENSION(:), INTENT(IN) :: scalv
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: y
    END SUBROUTINE solvde
END INTERFACE
INTERFACE
    SUBROUTINE sor(a,b,c,d,e,f,u,rjac)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: a,b,c,d,e,f
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
    REAL(DP), INTENT(IN) :: rjac
    END SUBROUTINE sor
END INTERFACE
INTERFACE
    SUBROUTINE sort(arr)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    END SUBROUTINE sort
END INTERFACE
INTERFACE
    SUBROUTINE sort2(arr,slave)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave
    END SUBROUTINE sort2
END INTERFACE
INTERFACE
    SUBROUTINE sort3(arr,slave1,slave2)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave1,slave2
    END SUBROUTINE sort3
END INTERFACE
INTERFACE
    SUBROUTINE sort_bypack(arr)
    USE nrtype
```

```
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    END SUBROUTINE sort_bypack
END INTERFACE
INTERFACE
    SUBROUTINE sort_byreshape(arr)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    END SUBROUTINE sort_byreshape
END INTERFACE
INTERFACE
    SUBROUTINE sort_heap(arr)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    END SUBROUTINE sort_heap
END INTERFACE
INTERFACE
    SUBROUTINE sort_pick(arr)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    END SUBROUTINE sort_pick
END INTERFACE
INTERFACE
    SUBROUTINE sort_radix(arr)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    END SUBROUTINE sort_radix
END INTERFACE
INTERFACE
    SUBROUTINE sort_shell(arr)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    END SUBROUTINE sort_shell
END INTERFACE
INTERFACE
    SUBROUTINE spctrm(p,k,ovrlap,unit,n_window)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: p
    INTEGER(I4B), INTENT(IN) :: k
    LOGICAL(LGT), INTENT(IN) :: ovrlap
    INTEGER(I4B), OPTIONAL, INTENT(IN) :: n_window,unit
    END SUBROUTINE spctrm
END INTERFACE
INTERFACE
    SUBROUTINE spear(data1,data2,d,zd,probd,rs,probrs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    REAL(SP), INTENT(OUT) :: d,zd,probd,rs,probrs
    END SUBROUTINE spear
END INTERFACE
INTERFACE sphbes
    SUBROUTINE sphbes_s(n,x,sj,sy,sjp,syp)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: sj,sy,sjp,syp
    END SUBROUTINE sphbes_s
    SUBROUTINE sphbes_v(n,x,sj,sy,sjp,syp)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: sj,sy,sjp,syp
    END SUBROUTINE sphbes_v
END INTERFACE
```

```
INTERFACE
    SUBROUTINE splie2(x1a,x2a,ya,y2a)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: y2a
    END SUBROUTINE splie2
END INTERFACE
INTERFACE
    FUNCTION splin2(x1a,x2a,ya,y2a,x1,x2)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya,y2a
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP) :: splin2
    END FUNCTION splin2
END INTERFACE
INTERFACE
    SUBROUTINE spline(x,y,yp1,ypn,y2)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), INTENT(IN) :: yp1,ypn
    REAL(SP), DIMENSION(:), INTENT(OUT) :: y2
    END SUBROUTINE spline
END INTERFACE
INTERFACE
    FUNCTION splint(xa,ya,y2a,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya,y2a
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: splint
    END FUNCTION splint
END INTERFACE
INTERFACE sprsax
    SUBROUTINE sprsax_dp(sa,x,b)
    USE nrtype
    TYPE(sprs2_dp), INTENT(IN) :: sa
    REAL(DP), DIMENSION (:), INTENT(IN) :: x
    REAL(DP), DIMENSION (:), INTENT(OUT) :: b
    END SUBROUTINE sprsax_dp

    SUBROUTINE sprsax_sp(sa,x,b)
    USE nrtype
    TYPE(sprs2_sp), INTENT(IN) :: sa
    REAL(SP), DIMENSION (:), INTENT(IN) :: x
    REAL(SP), DIMENSION (:), INTENT(OUT) :: b
    END SUBROUTINE sprsax_sp
END INTERFACE
INTERFACE sprsdiag
    SUBROUTINE sprsdiag_dp(sa,b)
    USE nrtype
    TYPE(sprs2_dp), INTENT(IN) :: sa
    REAL(DP), DIMENSION(:), INTENT(OUT) :: b
    END SUBROUTINE sprsdiag_dp

    SUBROUTINE sprsdiag_sp(sa,b)
    USE nrtype
    TYPE(sprs2_sp), INTENT(IN) :: sa
    REAL(SP), DIMENSION(:), INTENT(OUT) :: b
    END SUBROUTINE sprsdiag_sp
END INTERFACE
INTERFACE sprsin
    SUBROUTINE sprsin_sp(a,thresh,sa)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
```

```
    REAL(SP), INTENT(IN) :: thresh
    TYPE(sprs2_sp), INTENT(OUT) :: sa
    END SUBROUTINE sprsin_sp
    SUBROUTINE sprsin_dp(a,thresh,sa)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: a
    REAL(DP), INTENT(IN) :: thresh
    TYPE(sprs2_dp), INTENT(OUT) :: sa
    END SUBROUTINE sprsin_dp
END INTERFACE
INTERFACE
    SUBROUTINE sprstp(sa)
    USE nrtype
    TYPE(sprs2_sp), INTENT(INOUT) :: sa
    END SUBROUTINE sprstp
END INTERFACE
INTERFACE sprstx
    SUBROUTINE sprstx_dp(sa,x,b)
    USE nrtype
    TYPE(sprs2_dp), INTENT(IN) :: sa
    REAL(DP), DIMENSION (:), INTENT(IN) :: x
    REAL(DP), DIMENSION (:), INTENT(OUT) :: b
    END SUBROUTINE sprstx_dp
    SUBROUTINE sprstx_sp(sa,x,b)
    USE nrtype
    TYPE(sprs2_sp), INTENT(IN) :: sa
    REAL(SP), DIMENSION (:), INTENT(IN) :: x
    REAL(SP), DIMENSION (:), INTENT(OUT) :: b
    END SUBROUTINE sprstx_sp
END INTERFACE
INTERFACE
    SUBROUTINE stifbs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(OUT) :: hdid,hnext
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE stifbs
END INTERFACE
INTERFACE
    SUBROUTINE stiff(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(OUT) :: hdid,hnext
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
```

```
    END INTERFACE
    END SUBROUTINE stiff
END INTERFACE
INTERFACE
    SUBROUTINE stoerm(y,d2y,xs,htot,nstep,yout,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: y,d2y
    REAL(SP), INTENT(IN) :: xs,htot
    INTEGER(I4B), INTENT(IN) :: nstep
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE stoerm
END INTERFACE
INTERFACE svbksb
    SUBROUTINE svbksb_dp(u,w,v,b,x)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,v
    REAL(DP), DIMENSION(:), INTENT(IN) :: w,b
    REAL(DP), DIMENSION(:), INTENT(OUT) :: x
    END SUBROUTINE svbksb_dp

    SUBROUTINE svbksb_sp(u,w,v,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: u,v
    REAL(SP), DIMENSION(:), INTENT(IN) :: w,b
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x
    END SUBROUTINE svbksb_sp
END INTERFACE
INTERFACE svdcmp
    SUBROUTINE svdcmp_dp(a,w,v)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(DP), DIMENSION(:), INTENT(OUT) :: w
    REAL(DP), DIMENSION(:,:), INTENT(OUT) :: v
    END SUBROUTINE svdcmp_dp

    SUBROUTINE svdcmp_sp(a,w,v)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: w
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
    END SUBROUTINE svdcmp_sp
END INTERFACE
INTERFACE
    SUBROUTINE svdfit(x,y,sig,a,v,w,chisq,funcs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
    REAL(SP), DIMENSION(:), INTENT(OUT) :: a,w
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
    REAL(SP), INTENT(OUT) :: chisq
    INTERFACE
        FUNCTION funcs(x,n)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        INTEGER(I4B), INTENT(IN) :: n
        REAL(SP), DIMENSION(n) :: funcs
        END FUNCTION funcs
    END INTERFACE
```

```
        END SUBROUTINE svdfit
    END INTERFACE
    INTERFACE
        SUBROUTINE svdvar(v,w,cvm)
        USE nrtype
        REAL(SP), DIMENSION(:,:), INTENT(IN) :: v
        REAL(SP), DIMENSION(:), INTENT(IN) :: w
        REAL(SP), DIMENSION(:,:), INTENT(OUT) :: cvm
        END SUBROUTINE svdvar
    END INTERFACE
    INTERFACE
        FUNCTION toeplz(r,y)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: r,y
        REAL(SP), DIMENSION(size(y)) :: toeplz
        END FUNCTION toeplz
    END INTERFACE
    INTERFACE
        SUBROUTINE tptest(data1,data2,t,prob)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
        REAL(SP), INTENT(OUT) :: t,prob
        END SUBROUTINE tptest
    END INTERFACE
    INTERFACE
        SUBROUTINE tqli(d,e,z)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(INOUT) :: d,e
        REAL(SP), DIMENSION(:,:), OPTIONAL, INTENT(INOUT) :: z
        END SUBROUTINE tqli
    END INTERFACE
    INTERFACE
        SUBROUTINE trapzd(func,a,b,s,n)
        USE nrtype
        REAL(SP), INTENT(IN) :: a,b
        REAL(SP), INTENT(INOUT) :: s
        INTEGER(I4B), INTENT(IN) :: n
        INTERFACE
            FUNCTION func(x)
            USE nrtype
            REAL(SP), DIMENSION(:), INTENT(IN) :: x
            REAL(SP), DIMENSION(size(x)) :: func
            END FUNCTION func
        END INTERFACE
        END SUBROUTINE trapzd
    END INTERFACE
    INTERFACE
        SUBROUTINE tred2(a,d,e,novectors)
        USE nrtype
        REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
        REAL(SP), DIMENSION(:), INTENT(OUT) :: d,e
        LOGICAL(LGT), OPTIONAL, INTENT(IN) :: novectors
        END SUBROUTINE tred2
    END INTERFACE
!   On a purely serial machine, for greater efficiency, remove
!   the generic name tridag from the following interface,
!   and put it on the next one after that.
    INTERFACE tridag
        RECURSIVE SUBROUTINE tridag_par(a,b,c,r,u)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
        REAL(SP), DIMENSION(:), INTENT(OUT) :: u
        END SUBROUTINE tridag_par
    END INTERFACE
```

```
INTERFACE
    SUBROUTINE tridag_ser(a,b,c,r,u)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
    REAL(SP), DIMENSION(:), INTENT(OUT) :: u
    END SUBROUTINE tridag_ser
END INTERFACE
INTERFACE
    SUBROUTINE ttest(data1,data2,t,prob)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    REAL(SP), INTENT(OUT) :: t,prob
    END SUBROUTINE ttest
END INTERFACE
INTERFACE
    SUBROUTINE tutest(data1,data2,t,prob)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    REAL(SP), INTENT(OUT) :: t,prob
    END SUBROUTINE tutest
END INTERFACE
INTERFACE
    SUBROUTINE twofft(data1,data2,fft1,fft2)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: fft1,fft2
    END SUBROUTINE twofft
END INTERFACE
INTERFACE
    FUNCTION vander(x,q)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x,q
    REAL(DP), DIMENSION(size(x)) :: vander
    END FUNCTION vander
END INTERFACE
INTERFACE
    SUBROUTINE vegas(region,func,init,ncall,itmx,nprn,tgral,sd,chi2a)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: region
    INTEGER(I4B), INTENT(IN) :: init,ncall,itmx,nprn
    REAL(SP), INTENT(OUT) :: tgral,sd,chi2a
    INTERFACE
        FUNCTION func(pt,wgt)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(IN) :: pt
        REAL(SP), INTENT(IN) :: wgt
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE vegas
END INTERFACE
INTERFACE
    SUBROUTINE voltra(t0,h,t,f,g,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: t0,h
    REAL(SP), DIMENSION(:), INTENT(OUT) :: t
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: f
    INTERFACE
        FUNCTION g(t)
        USE nrtype
        REAL(SP), INTENT(IN) :: t
        REAL(SP), DIMENSION(:), POINTER :: g
        END FUNCTION g
```

```
        FUNCTION ak(t,s)
        USE nrtype
        REAL(SP), INTENT(IN) :: t,s
        REAL(SP), DIMENSION(:,:), POINTER :: ak
        END FUNCTION ak
    END INTERFACE
    END SUBROUTINE voltra
END INTERFACE
INTERFACE
    SUBROUTINE wt1(a,isign,wtstep)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    INTERFACE
        SUBROUTINE wtstep(a,isign)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
        INTEGER(I4B), INTENT(IN) :: isign
        END SUBROUTINE wtstep
    END INTERFACE
    END SUBROUTINE wt1
END INTERFACE
INTERFACE
    SUBROUTINE wtn(a,nn,isign,wtstep)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
    INTEGER(I4B), INTENT(IN) :: isign
    INTERFACE
        SUBROUTINE wtstep(a,isign)
        USE nrtype
        REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
        INTEGER(I4B), INTENT(IN) :: isign
        END SUBROUTINE wtstep
    END INTERFACE
    END SUBROUTINE wtn
END INTERFACE
INTERFACE
    FUNCTION wwghts(n,h,kermom)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: h
    REAL(SP), DIMENSION(n) :: wwghts
    INTERFACE
        FUNCTION kermom(y,m)
        USE nrtype
        REAL(DP), INTENT(IN) :: y
        INTEGER(I4B), INTENT(IN) :: m
        REAL(DP), DIMENSION(m) :: kermom
        END FUNCTION kermom
    END INTERFACE
    END FUNCTION wwghts
END INTERFACE
INTERFACE
    SUBROUTINE zbrac(func,x1,x2,succes)
    USE nrtype
    REAL(SP), INTENT(INOUT) :: x1,x2
    LOGICAL(LGT), INTENT(OUT) :: succes
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
```

```
    END INTERFACE
    END SUBROUTINE zbrac
END INTERFACE
INTERFACE
    SUBROUTINE zbrak(func,x1,x2,n,xb1,xb2,nb)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B), INTENT(OUT) :: nb
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP), DIMENSION(:), POINTER :: xb1,xb2
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END SUBROUTINE zbrak
END INTERFACE
INTERFACE
    FUNCTION zbrent(func,x1,x2,tol)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,tol
    REAL(SP) :: zbrent
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION zbrent
END INTERFACE
INTERFACE
    SUBROUTINE zrhqr(a,rtr,rti)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: rtr,rti
    END SUBROUTINE zrhqr
END INTERFACE
INTERFACE
    FUNCTION zriddr(func,x1,x2,xacc)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: zriddr
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP) :: func
        END FUNCTION func
    END INTERFACE
    END FUNCTION zriddr
END INTERFACE
INTERFACE
    SUBROUTINE zroots(a,roots,polish)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
    COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: roots
    LOGICAL(LGT), INTENT(IN) :: polish
    END SUBROUTINE zroots
END INTERFACE
END MODULE nr
```

C3. Index of Programs and Dependencies

The following table lists, in alphabetical order, all the routines in Volume 2 of *Numerical Recipes*. When a routine requires subsidiary routines, either from this book or else user-supplied, the full dependency tree is shown: A routine calls directly all routines to which it is connected by a solid line in the column immediately to its right; it calls indirectly the connected routines in all columns to its right. Typographical conventions: Routines from this book are in typewriter font (e.g., `eulsum`, *`gammln`*). The smaller, slanted font is used for the second and subsequent occurrences of a routine in a single dependency tree. (When you are getting routines from the *Numerical Recipes* machine-readable media or hypertext archives, you need specify names only in the larger, upright font.) User-supplied routines are indicated by the use of text font and square brackets, e.g., [funcv]. Consult the text for individual specifications of these routines. The right-hand side of the table lists chapter and page numbers for each program.

`airy` — `bessik`, `bessjy` — `beschb` — `chebev`	B6 (p. 1121)
`amebsa` — `ran1` — `ran_state`; [func]	B10 (p. 1222)
`amoeba` — [func]	B10 (p. 1208)
`anneal` — `ran1` — `ran_state`	B10 (p. 1219)
`arcmak`	B20 (p. 1349)
`arcode` — `arcmak`	B20 (p. 1350)
`avevar`	B14 (p. 1270)
`badluk` — `julday`, `flmoon`	B1 (p. 1011)
`balanc`	B11 (p. 1230)
`banbks`	B2 (p. 1021)
`bandec`	B2 (p. 1020)
`banmul`	B2 (p. 1019)
`bcucof`	B3 (p. 1049)
`bcuint` — `bcucof`	B3 (p. 1050)
`beschb` — `chebev`	B6 (p. 1118)

`bessi` — `bessi0` . . . . . B6 (p. 1114)

`bessi0` . . . . . B6 (p. 1109)

`bessi1` . . . . . B6 (p. 1111)

`bessik` — `beschb` — `chebev` . . . . . B6 (p. 1118)

`bessj` — `bessj0` . . . . . B6 (p. 1106)
`bessj1`

`bessj0` . . . . . B6 (p. 1101)

`bessj1` . . . . . B6 (p. 1103)

`bessjy` — `beschb` — `chebev` . . . . . B6 (p. 1115)

`bessk` — `bessk0` — `bessi0` . . . . . B6 (p. 1113)
`bessk1` — `bessi1`

`bessk0` — `bessi0` . . . . . B6 (p. 1110)

`bessk1` — `bessi1` . . . . . B6 (p. 1112)

`bessy` — `bessy1` — `bessj1` . . . . . B6 (p. 1105)
`bessy0` — `bessj0`

`bessy0` — `bessj0` . . . . . B6 (p. 1102)

`bessy1` — `bessj1` . . . . . B6 (p. 1104)

`beta` — `gammln` . . . . . B6 (p. 1089)

`betacf` . . . . . B6 (p. 1099)

`betai` — `gammln` . . . . . B6 (p. 1098)
`betacf`

`bico` — `factln` — `gammln` . . . . . B6 (p. 1087)

`bnldev` — `ran1` — `ran_state` . . . . . B7 (p. 1155)
`gammln`

`brent` — [func] . . . . . B10 (p. 1204)

`broydn` — `fmin` . . . . . B9 (p. 1199)
`fdjac` — [funcv]
`qrdcmp`
`qrupdt` — `rotate`
`pythag`
`rsolv`
`lnsrch` — *`fmin`* — [funcv]

`bsstep` — `mmid` — [derivs] . . . . . B16 (p. 1303)
`pzextr`

`caldat` . . . . . B1 (p. 1013)

`chder` . . . . . B5 (p. 1077)

`chebev` . . . . . B5 (p. 1076)

`chebft` — [func] . . . . . B5 (p. 1076)

`chebpc` . . . . . B5 (p. 1078)

`chint` . . . . . B5 (p. 1078)

`chixy` . . . . . B15 (p. 1287)

choldc . . . . . B2 (p. 1038)

cholsl . . . . . B2 (p. 1039)

chsone — gammq — gser — gcf — gammln . . . . . B14 (p. 1272)

chstwo — gammq — gser — gcf — gammln . . . . . B14 (p. 1272)

cisi . . . . . B6 (p. 1125)

cntab1 — gammq — gser — gcf — gammln . . . . . B14 (p. 1275)

cntab2 . . . . . B14 (p. 1275)

convlv — realft — four1 — fourrow . . . . . B13 (p. 1253)

correl — realft — four1 — fourrow . . . . . B13 (p. 1254)

cosft1 — realft — four1 — fourrow . . . . . B12 (p. 1245)

cosft2 — realft — four1 — fourrow . . . . . B12 (p. 1246)

covsrt . . . . . B15 (p. 1289)

cyclic — tridag . . . . . B2 (p. 1030)

daub4 . . . . . B13 (p. 1264)

dawson . . . . . B6 (p. 1127)

dbrent — [func] . . . . . B10 (p. 1205)
dbrent — [dfunc]

ddpoly . . . . . B5 (p. 1071)

decchk . . . . . B20 (p. 1345)

dfpmin — [func] . . . . . B10 (p. 1215)
dfpmin — [dfunc]
dfpmin — lnsrch — [func]

dfridr — [func] . . . . . B5 (p. 1075)

dftcor . . . . . B13 (p. 1261)

dftint — [func] . . . . . B13 (p. 1263)
dftint — realft — four1 — fourrow
dftint — polint
dftint — dftcor

difeq . . . . . B17 (p. 1320)

dlinmin — mnbrak . . . . . B10 (p. 1212)
dlinmin — dbrent — [func]
dlinmin — dbrent — [dfunc]

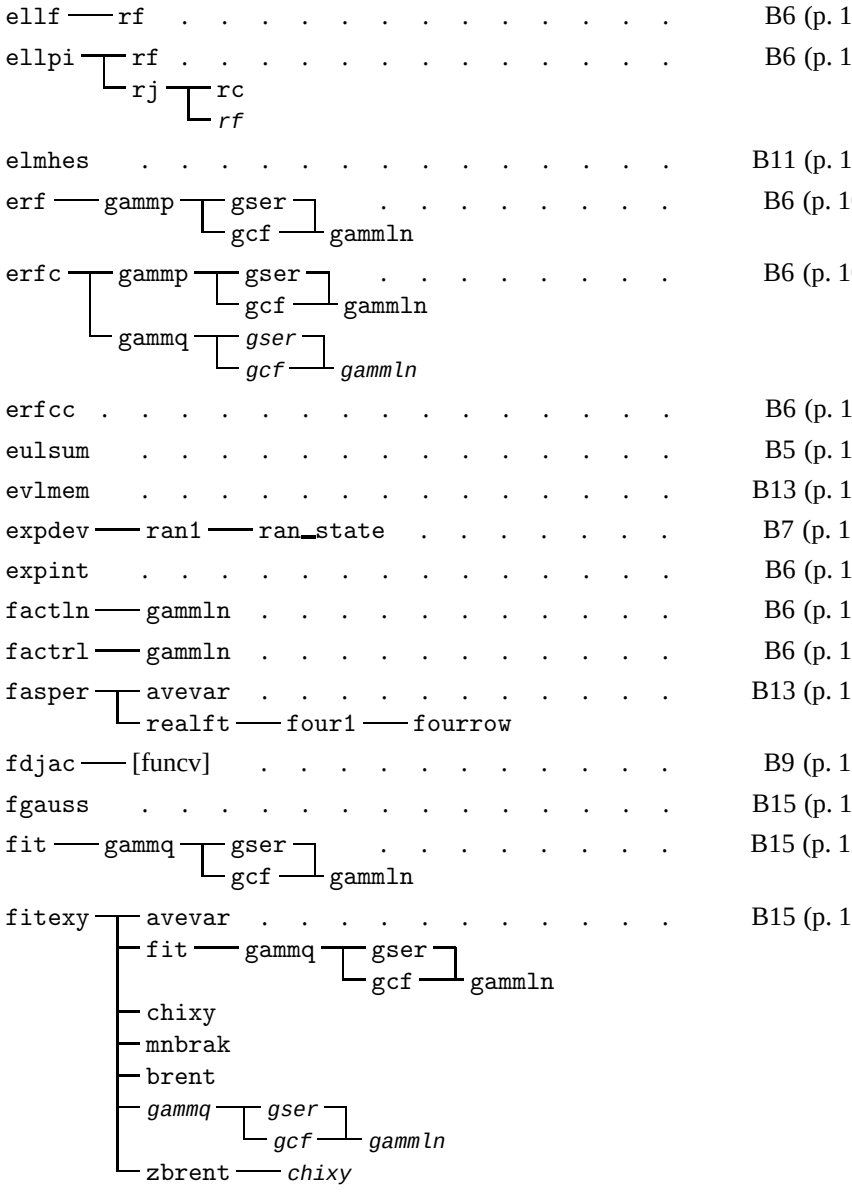
eclass . . . . . B8 (p. 1180)

eclazz — [equiv] . . . . . B8 (p. 1180)

ei . . . . . B6 (p. 1097)

eigsrt . . . . . B11 (p. 1227)

elle — rf . . . . . B6 (p. 1136)
elle — rd


ellf rf B6 (p. 1135)
ellpi rf B6 (p. 1136)
rj rc
rf
elmhes B11 (p. 1231)
erf gammp gser B6 (p. 1094)
gcf gammln
erfc gammp gser B6 (p. 1094)
gcf gammln
gammq gser
gcf gammln
erfcc B6 (p. 1095)
eulsum B5 (p. 1070)
evlmem B13 (p. 1258)
expdev ran1 ran_state B7 (p. 1151)
expint B6 (p. 1096)
factln gammln B6 (p. 1088)
factrl gammln B6 (p. 1086)
fasper avevar B13 (p. 1259)
realft four1 fourrow
fdjac [funcv] B9 (p. 1197)
fgauss B15 (p. 1294)
fit gammq gser B15 (p. 1285)
gcf gammln
fitexy avevar B15 (p. 1286)
fit gammq gser
gcf gammln
chixy
mnbrak
brent
gammq gser
gcf gammln
zbrent chixy
fixrts zroots laguer B13 (p. 1257)
indexx
fleg B15 (p. 1291)
flmoon B1 (p. 1010)
fmin [funcv] B9 (p. 1198)
four1 fourrow B12 (p. 1239)
four1_alt fourcol B12 (p. 1240)
four1_gather B12 (p. 1250)
four2 fourrow B12 (p. 1241)

four2_alt — fourcol . . . . . B12 (p. 1242)
four3 — fourrow_3d . . . . . B12 (p. 1246)
four3_alt — fourcol_3d . . . . . B12 (p. 1247)
fourcol . . . . . B12 (p. 1237)
fourcol_3d . . . . . B12 (p. 1238)
fourn_gather . . . . . B12 (p. 1251)
fourrow . . . . . B12 (p. 1235)
fourrow_3d . . . . . B12 (p. 1236)
fpoly . . . . . B15 (p. 1291)
fred2 — gauleg . . . . . B18 (p. 1325)
  [ak]
  [g]
  ludcmp
  lubksb
fredex — quadmx — wwghts — kermom . . . . B18 (p. 1331)
  ludcmp
  lubksb
fredin — [ak] . . . . . B18 (p. 1326)
  [g]
frenel . . . . . B6 (p. 1123)
frprmn — [func] . . . . . B10 (p. 1214)
  [dfunc]
  linmin — mnbrak
    brent — [func]
ftest — avevar . . . . . B14 (p. 1271)
  betai — gammln
    betacf
gamdev — ran1 — ran_state . . . . . B7 (p. 1153)
gammln . . . . . B6 (p. 1085)
gammp — gser . . . . . B6 (p. 1089)
  gcf — gammln
gammq — gser . . . . . B6 (p. 1090)
  gcf — gammln
gasdev — ran1 — ran_state . . . . . B7 (p. 1152)
gaucof — tqli — pythag . . . . . B4 (p. 1064)
  eigsrt
gauher . . . . . B4 (p. 1062)
gaujac — gammln . . . . . B4 (p. 1063)
gaulag — gammln . . . . . B4 (p. 1060)
gauleg . . . . . B4 (p. 1059)
gaussj . . . . . B2 (p. 1014)
gcf — gammln . . . . . B6 (p. 1092)

golden — [func] . . . . . B10 (p. 1202)

gser — gammln . . . . . B6 (p. 1090)

hqr . . . . . B11 (p. 1232)

hufdec — hufmak . . . . . B20 (p. 1349)

hufenc — hufmak . . . . . B20 (p. 1348)

hufmak . . . . . B20 (p. 1346)

hunt . . . . . B3 (p. 1046)

hypdrv . . . . . B6 (p. 1139)

hypgeo — hypser . . . . . B6 (p. 1138)
hypgeo — odeint — bsstep — mmid
hypgeo — odeint — bsstep — pzextr
hypgeo — odeint — hypdrv

hypser . . . . . B6 (p. 1139)

icrc . . . . . B20 (p. 1344)

igray . . . . . B20 (p. 1344)

index_bypack . . . . . B8 (p. 1176)

indexx . . . . . B8 (p. 1173)

interp . . . . . B19 (p. 1337)

irbit1 . . . . . B7 (p. 1159)

irbit2 . . . . . B7 (p. 1160)

jacobi . . . . . B11 (p. 1225)

jacobn . . . . . B16 (p. 1309)

julday . . . . . B1 (p. 1011)

kendl1 — erfcc . . . . . B14 (p. 1279)

kendl2 — erfcc . . . . . B14 (p. 1279)

kermom . . . . . B18 (p. 1329)

ks2d1s — quadct . . . . . B14 (p. 1281)
ks2d1s — quadvl
ks2d1s — pearsn — betai — gammln
ks2d1s — pearsn — betai — betacf
ks2d1s — probks

ks2d2s — quadct . . . . . B14 (p. 1283)
ks2d2s — pearsn — betai — gammln
ks2d2s — pearsn — betai — betacf
ks2d2s — probks

ksone — sort . . . . . B14 (p. 1273)
ksone — [func]
ksone — probks

kstwo — sort2 . . . . . B14 (p. 1273)
kstwo — probks

laguer . . . . . B9 (p. 1191)

lfit — [funcs] . . . . . B15 (p. 1288)
  gaussj
  covsrt

linbcg — atimes . . . . . B2 (p. 1034)
  snrm
  asolve

linmin — mnbrak — [func] . . . . . B10 (p. 1211)
  brent — [func]

lnsrch — [func] . . . . . B9 (p. 1195)

locate . . . . . B3 (p. 1045)

lop . . . . . B19 (p. 1342)

lubksb . . . . . B2 (p. 1017)

ludcmp . . . . . B2 (p. 1016)

machar . . . . . B20 (p. 1343)

medfit — select . . . . . B15 (p. 1294)

memcof . . . . . B13 (p. 1256)

mgfas — rstrct . . . . . B19 (p. 1339)
  slvsm2
  interp
  relax2
  lop

mglin — rstrct . . . . . B19 (p. 1334)
  slvsml
  interp
  relax
  resid

midexp — [funk] . . . . . B4 (p. 1058)

midinf — [funk] . . . . . B4 (p. 1056)

midpnt — [func] . . . . . B4 (p. 1054)

midsql — [funk] . . . . . B4 (p. 1057)

midsqu — [funk] . . . . . B4 (p. 1057)

miser — ran1 — ran_state . . . . . B7 (p. 1164)
  [func]

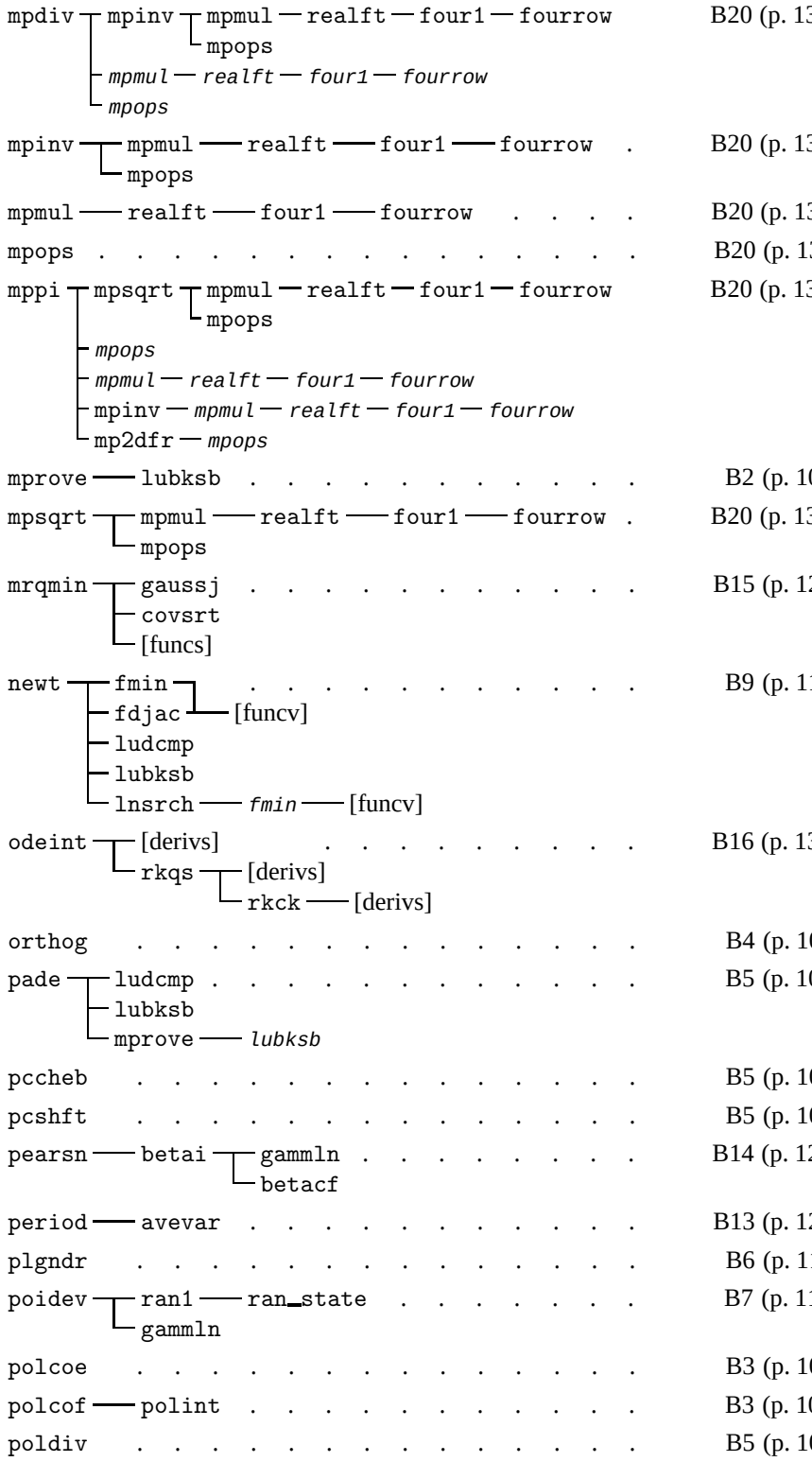
mmid — [derivs] . . . . . B16 (p. 1302)

mnbrak — [func] . . . . . B10 (p. 1201)

mnewt — [usrfun] . . . . . B9 (p. 1194)
  ludcmp
  lubksb

moment . . . . . B14 (p. 1269)

mp2dfr — mpops . . . . . B20 (p. 1357)


mpdiv mpinv mpmul realft four1 fourrow B20 (p. 1356)
mpops
mpmul realft four1 fourrow
mpops
mpinv mpmul realft four1 fourrow B20 (p. 1355)
mpops
mpmul realft four1 fourrow B20 (p. 1354)
mpops B20 (p. 1352)
mppi mpsqrt mpmul realft four1 fourrow B20 (p. 1357)
mpops
mpops
mpmul realft four1 fourrow
mpinv mpmul realft four1 fourrow
mp2dfr mpops
mprove lubksb B2 (p. 1022)
mpsqrt mpmul realft four1 fourrow B20 (p. 1356)
mpops
mrqmin gaussj B15 (p. 1292)
covsrt
[funcs]
newt fmin B9 (p. 1196)
fdjac [funcv]
ludcmp
lubksb
lnsrch fmin [funcv]
odeint [derivs] B16 (p. 1300)
rkqs [derivs]
rkck [derivs]
orthog B4 (p. 1064)
pade ludcmp B5 (p. 1080)
lubksb
mprove lubksb
pccheb B5 (p. 1080)
pcshft B5 (p. 1079)
pearsn betai gammln B14 (p. 1276)
betacf
period avevar B13 (p. 1258)
plgndr B6 (p. 1122)
poidev ran1 ran_state B7 (p. 1154)
gammln
polcoe B3 (p. 1047)
polcof polint B3 (p. 1048)
poldiv B5 (p. 1072)

polin2 — polint . . . . . B3 (p. 1049)

polint . . . . . B3 (p. 1043)

powell — [func] . . . . . B10 (p. 1210)
linmin — mnbrak
brent — [func]

predic . . . . . B13 (p. 1257)

probks . . . . . B14 (p. 1274)

psdes . . . . . B7 (p. 1156)

pwt — pwtset . . . . . B13 (p. 1266)

pwtset . . . . . B13 (p. 1265)

pythag . . . . . B2 (p. 1029)

pzextr . . . . . B16 (p. 1305)

qrdcmp . . . . . B2 (p. 1039)

qromb — trapzd — [func] . . . . . B4 (p. 1054)
polint

qromo — midpnt — [func] . . . . . B4 (p. 1055)
polint

qroot — poldiv . . . . . B9 (p. 1193)

qrsolv — rsolv . . . . . B2 (p. 1040)

qrupdt — rotate . . . . . B2 (p. 1041)
pythag

qsimp — trapzd — [func] . . . . . B4 (p. 1053)

qtrap — trapzd — [func] . . . . . B4 (p. 1053)

quad3d — polint . . . . . B4 (p. 1065)
[func]
[y1]
[y2]
[z1]
[z2]

quadct . . . . . B14 (p. 1282)

quadmx — wwghts — kermom . . . . . B18 (p. 1330)

quadvl . . . . . B14 (p. 1282)

ran . . . . . B7 (p. 1142)

ran0 — ran_state . . . . . B7 (p. 1148)

ran1 — ran_state . . . . . B7 (p. 1149)

ran2 — ran_state . . . . . B7 (p. 1150)

ran3 — ran_state . . . . . B7 (p. 1158)

ran_state . . . . . B7 (p. 1144)

rank . . . . . B8 (p. 1176)

ratint . . . . . B3 (p. 1043)

ratlsq ┬ [func] . . . . . B5 (p. 1081)
       ├ svdcmp ── pythag
       ├ svbksb
       └ ratval

ratval . . . . . B5 (p. 1072)

rc . . . . . B6 (p. 1134)

rd . . . . . B6 (p. 1130)

realft ── four1 ── fourrow . . . . . B12 (p. 1243)

recur1 . . . . . B5 (p. 1073)

recur2 . . . . . B5 (p. 1074)

relax . . . . . B19 (p. 1338)

relax2 . . . . . B19 (p. 1341)

resid . . . . . B19 (p. 1338)

rf . . . . . B6 (p. 1128)

rj ┬ rc . . . . . B6 (p. 1131)
   └ rf

rk4 ── [derivs] . . . . . B16 (p. 1297)

rkck ── [derivs] . . . . . B16 (p. 1299)

rkdumb ┬ [derivs] . . . . . B16 (p. 1297)
       └ rk4 ── [derivs]

rkqs ── rkck ── [derivs] . . . . . B16 (p. 1298)

rlft2 ── four2 ── fourrow . . . . . B12 (p. 1248)

rlft3 ── four3 ── fourrow_3d . . . . . B12 (p. 1249)

rotate . . . . . B2 (p. 1041)

rsolv . . . . . B2 (p. 1040)

rstrct . . . . . B19 (p. 1337)

rtbis ── [func] . . . . . B9 (p. 1184)

rtflsp ── [func] . . . . . B9 (p. 1185)

rtnewt ── [funcd] . . . . . B9 (p. 1189)

rtsafe ── [funcd] . . . . . B9 (p. 1190)

rtsec ── [func] . . . . . B9 (p. 1186)

rzextr . . . . . B16 (p. 1306)

savgol ┬ ludcmp . . . . . B14 (p. 1283)
       └ lubksb

scrsho ── [func] . . . . . B9 (p. 1182)

select . . . . . B8 (p. 1177)

select_bypack . . . . . B8 (p. 1178)

select_heap ── sort . . . . . B8 (p. 1179)

select_inplace ── select . . . . . B8 (p. 1178)

```
sfroid ─┬─ plgndr . . . . . B17 (p. 1319)
        └─ solvde ── difeq
```

```
shoot ─┬─ [load] . . . . . B17 (p. 1314)
       ├─ odeint ─┬─ [derivs]
       │          └─ rkqs ── rkck ── [derivs]
       └─ [score]
```

```
shootf ─┬─ [load1] . . . . . B17 (p. 1315)
        ├─ odeint ─┬─ [derivs]
        │          └─ rkqs ── rkck ── [derivs]
        ├─ [score]
        └─ [load2]
```

```
simplx . . . . . B10 (p. 1216)
```

```
simpr ─┬─ ludcmp . . . . . B16 (p. 1310)
       ├─ lubksb
       └─ [derivs]
```

```
sinft ── realft ── four1 ── fourrow . . . . B12 (p. 1245)
```

```
slvsm2 . . . . . B19 (p. 1342)
```

```
slvsml . . . . . B19 (p. 1337)
```

```
sncndn . . . . . B6 (p. 1137)
```

```
snrm . . . . . B2 (p. 1036)
```

```
sobseq . . . . . B7 (p. 1160)
```

```
solvde ── difeq . . . . . B17 (p. 1316)
```

```
sor . . . . . B19 (p. 1332)
```

```
sort . . . . . B8 (p. 1169)
```

```
sort2 ── indexx . . . . . B8 (p. 1170)
```

```
sort3 ── indexx . . . . . B8 (p. 1175)
```

```
sort_bypack . . . . . B8 (p. 1171)
```

```
sort_byreshape . . . . . B8 (p. 1168)
```

```
sort_heap . . . . . B8 (p. 1171)
```

```
sort_pick . . . . . B8 (p. 1167)
```

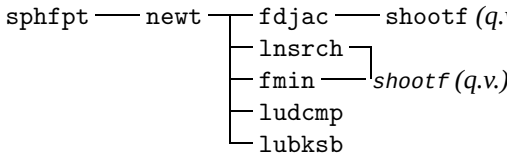
```
sort_radix . . . . . B8 (p. 1172)
```

```
sort_shell . . . . . B8 (p. 1168)
```

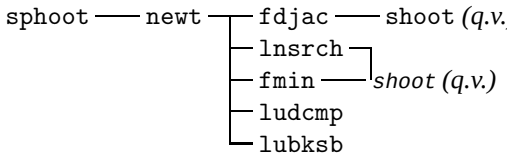
```
spctrm ── four1 ── fourrow . . . . . B13 (p. 1254)
```

```
spear ─┬─ sort2 . . . . . B14 (p. 1277)
       ├─ erfcc
       └─ betai ─┬─ gammln
                 └─ betacf
```

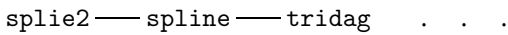
```
sphbes ── bessjy ── beschb ── chebev . . . . B6 (p. 1121)
```



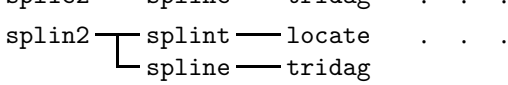
B17 (p. 1322)



B17 (p. 1321)



B3 (p. 1050)



B3 (p. 1051)

B3 (p. 1044)

B3 (p. 1045)

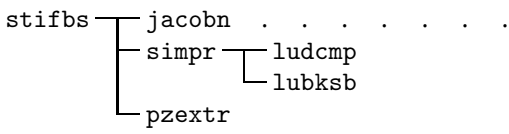
sprsax . . . . . B2 (p. 1032)

sprsdiag . . . . . B2 (p. 1033)

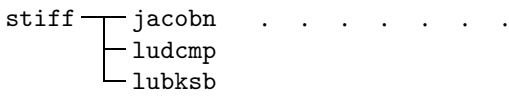
sprsin . . . . . B2 (p. 1031)

sprstp . . . . . B2 (p. 1033)

sprstx . . . . . B2 (p. 1032)



B16 (p. 1311)

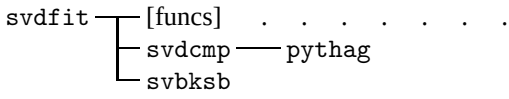


B16 (p. 1308)

B16 (p. 1307)

svbksb . . . . . B2 (p. 1022)

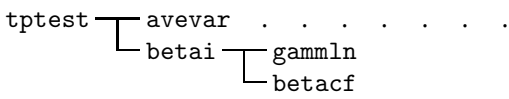
B2 (p. 1023)



B15 (p. 1290)

svdvar . . . . . B15 (p. 1290)

toeplz . . . . . B2 (p. 1038)



B14 (p. 1271)

B11 (p. 1228)

B4 (p. 1052)

tred2 . . . . . B11 (p. 1227)

tridag . . . . . B2 (p. 1018)

ttest
- avevar . . . . . B14 (p. 1269)
- betai
  - gammln
  - betacf

tutest
- avevar . . . . . B14 (p. 1270)
- betai
  - gammln
  - betacf

twofft — four1 — fourrow . . . . . B12 (p. 1242)

vander . . . . . B2 (p. 1037)

vegas
- ran1 — ran_state . . . . . B7 (p. 1161)
- [func]

voltra
- [g] . . . . . B18 (p. 1326)
- [ak]
- ludcmp
- lubksb

wt1 — daub4 . . . . . B13 (p. 1264)

wtn — daub4 . . . . . B13 (p. 1267)

wwghts — kermom . . . . . B18 (p. 1328)

zbrac — [func] . . . . . B9 (p. 1183)

zbrak — [func] . . . . . B9 (p. 1184)

zbrent — [func] . . . . . B9 (p. 1188)

zrhqr
- balanc . . . . . B9 (p. 1193)
- hqr
- indexx

zriddr — [func] . . . . . B9 (p. 1187)

zroots
- laguer . . . . . B9 (p. 1192)
- indexx

Chapter B2. Solution of Linear Algebraic Equations

```
SUBROUTINE gaussj(a,b)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerand,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
```

Linear equation solution by Gauss-Jordan elimination, equation (2.1.1). a is an $N \times N$ input coefficient matrix. b is an $N \times M$ input matrix containing M right-hand-side vectors. On output, a is replaced by its matrix inverse, and b is replaced by the corresponding set of solution vectors.

```
INTEGER(I4B), DIMENSION(size(a,1)) :: ipiv,indxr,indxc
```

These arrays are used for bookkeeping on the pivoting.

```
LOGICAL(LGT), DIMENSION(size(a,1)) :: lpiv
REAL(SP) :: pivinv
REAL(SP), DIMENSION(size(a,1)) :: dumc
INTEGER(I4B), TARGET :: irc(2)
INTEGER(I4B) :: i,l,n
INTEGER(I4B), POINTER :: irow,icol
n=assert_eq(size(a,1),size(a,2),size(b,1),'gaussj')
irow => irc(1)
icol => irc(2)
ipiv=0
do i=1,n                          Main loop over columns to be reduced.
    lpiv = (ipiv == 0)            Begin search for a pivot element.
    irc=maxloc(abs(a),outerand(lpiv,lpiv))
    ipiv(icol)=ipiv(icol)+1
    if (ipiv(icol) > 1) call nrerror('gaussj: singular matrix (1)')
```

We now have the pivot element, so we interchange rows, if needed, to put the pivot element on the diagonal. The columns are not physically interchanged, only relabeled: indxc(i), the column of the ith pivot element, is the ith column that is reduced, while indxr(i) is the row in which that pivot element was originally located. If indxr(i) $\neq$ indxc(i) there is an implied column interchange. With this form of bookkeeping, the solution b's will end up in the correct order, and the inverse matrix will be scrambled by columns.

```
    if (irow /= icol) then
        call swap(a(irow,:),a(icol,:))
        call swap(b(irow,:),b(icol,:))
    end if
    indxr(i)=irow                 We are now ready to divide the pivot row by the pivot
    indxc(i)=icol                     element, located at irow and icol.
    if (a(icol,icol) == 0.0) &
        call nrerror('gaussj: singular matrix (2)')
    pivinv=1.0_sp/a(icol,icol)
    a(icol,icol)=1.0
    a(icol,:)=a(icol,:)*pivinv
    b(icol,:)=b(icol,:)*pivinv
    dumc=a(:,icol)                Next, we reduce the rows, except for the pivot one, of
    a(:,icol)=0.0                     course.
```

```
    a(icol,icol)=pivinv
    a(1:icol-1,:)=a(1:icol-1,:)-outerprod(dumc(1:icol-1),a(icol,:))
    b(1:icol-1,:)=b(1:icol-1,:)-outerprod(dumc(1:icol-1),b(icol,:))
    a(icol+1:,:)=a(icol+1:,:)-outerprod(dumc(icol+1:),a(icol,:))
    b(icol+1:,:)=b(icol+1:,:)-outerprod(dumc(icol+1:),b(icol,:))
end do
```

It only remains to unscramble the solution in view of the column interchanges. We do this by interchanging pairs of columns in the reverse order that the permutation was built up.

```
do l=n,1,-1
    call swap(a(:,indxr(l)),a(:,indxc(l)))
end do
END SUBROUTINE gaussj
```

f90 `irow => irc(1) ... icol => irc(2)` The `maxloc` intrinsic returns the location of the maximum value of an array as an integer array, in this case of size 2. Pre-pointing pointer variables to components of the array that will be thus set makes possible convenient references to the desired row and column positions.

`irc=maxloc(abs(a),outerand(lpiv,lpiv))` The combination of `maxloc` and one of the `outer...` routines from `nrutil` allows for a very concise formulation. If this task is done with loops, it becomes the ungainly "flying vee,"

```
aa=0.0
do i=1,n
   if (lpiv(i)) then
      do j=1,n
         if (lpiv(j)) then
            if (abs(a(i,j)) > aa) then
               aa=abs(a(i,j))
               irow=i
               icol=j
            endif
         endif
      end do
   end do
end do
```

`call swap(a(irow,:),a(icol,:))` The `swap` routine (in `nrutil`) is concise and convenient. Fortran 90's ability to overload multiple routines onto a single name is vital here: Much of the convenience would vanish if we had to remember variant routine names for each variable type and rank of object that might be swapped.

Even better, here, than overloading would be if Fortran 90 allowed user-written *elemental* procedures (procedures with unspecified or arbitrary rank and shape), like the intrinsic elemental procedures built into the language. Fortran 95 will, but Fortran 90 doesn't.

One quick (if superficial) test for how much parallelism is achieved in a Fortran 90 routine is to count its do-loops, and compare that number to the number of do-loops in the Fortran 77 version of the same routine. Here, in `gaussj`, 13 do-loops are reduced to 2.

```
a(1:icol-1,:)=... b(1:icol-1,:)=...
a(icol+1:,:)=...  b(icol+1:,:)=...
```

Here the same operation is applied to every row of a, and to every row of b, *except* row number icol. On a massively multiprocessor (MMP) machine it would be better to use a logical mask and do all of a in a single statement, all of b in another one. For a small-scale parallel (SSP) machine, the lines as written should saturate the machine's concurrency, and they avoid the additional overhead of testing the mask.

This would be a good place to point out, however, that linear algebra routines written in Fortran 90 are likely *never* to be competitive with the hand-coded library routines that are generally supplied as part of MMP programming environments. If you are using our routines instead of library routines written specifically for your architecture, you are wasting cycles!

⋆ ⋆ ⋆

```
SUBROUTINE ludcmp(a,indx,d)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,nrerror,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
REAL(SP), INTENT(OUT) :: d
```

Given an $N \times N$ input matrix a, this routine replaces it by the LU decomposition of a rowwise permutation of itself. On output, a is arranged as in equation (2.3.14); indx is an output vector of length N that records the row permutation effected by the partial pivoting; d is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with lubksb to solve linear equations or invert a matrix.

```
REAL(SP), DIMENSION(size(a,1)) :: vv          vv stores the implicit scaling of each row.
REAL(SP), PARAMETER :: TINY=1.0e-20_sp        A small number.
INTEGER(I4B) :: j,n,imax
n=assert_eq(size(a,1),size(a,2),size(indx),'ludcmp')
d=1.0                                         No row interchanges yet.
vv=maxval(abs(a),dim=2)                       Loop over rows to get the implicit scaling
if (any(vv == 0.0)) call nrerror('singular matrix in ludcmp')      information.
```

There is a row of zeros.

```
vv=1.0_sp/vv                                  Save the scaling.
do j=1,n
    imax=(j-1)+imaxloc(vv(j:n)*abs(a(j:n,j)))        Find the pivot row.
    if (j /= imax) then                       Do we need to interchange rows?
        call swap(a(imax,:),a(j,:))           Yes, do so...
        d=-d                                  ...and change the parity of d.
        vv(imax)=vv(j)                        Also interchange the scale factor.
    end if
    indx(j)=imax
    if (a(j,j) == 0.0) a(j,j)=TINY
```

If the pivot element is zero the matrix is singular (at least to the precision of the algorithm). For some applications on singular matrices, it is desirable to substitute TINY for zero.

```
    a(j+1:n,j)=a(j+1:n,j)/a(j,j)              Divide by the pivot element.
    a(j+1:n,j+1:n)=a(j+1:n,j+1:n)-outerprod(a(j+1:n,j),a(j,j+1:n))
```

Reduce remaining submatrix.

```
end do
END SUBROUTINE ludcmp
```

f90 vv=maxval(abs(a),dim=2) A single statement finds the maximum absolute value in each row. Fortran 90 intrinsics like maxval generally "do their thing" *in* the dimension specified by *dim* and return a result with a shape corresponding to the *other* dimensions. Thus, here, vv's size is that of the *first* dimension of a.

`imax=(j-1)+imaxloc(vv(j:n)*abs(a(j:n,j))` Here we see why the `nrutil` routine `imaxloc` is handy: We want the index, in the range `1:n` of a quantity to be searched for only in the limited range `j:n`. Using `imaxloc`, we just add back the proper offset of `j-1`. (Using only Fortran 90 intrinsics, we could write `imax=(j-1)+sum(maxloc(vv(j:n)*abs(a(j:n,j))))`, but the use of `sum` just to turn an array of length 1 into a scalar seems sufficiently confusing as to be avoided.)

`a(j+1:n,j+1:n)=a(j+1:n,j+1:n)-outerprod(a(j+1:n,j),a(j,j+1:n))` The Fortran 77 version of `ludcmp`, using Crout's algorithm for the reduction, does not parallelize well: The elements are updated by $O(N^2)$ separate dot product operations in a particular order. Here we use a slightly different reduction, termed "outer product Gaussian elimination" by Golub and Van Loan [1], that requires just N steps of matrix-parallel reduction. (See their §3.2.3 and §3.2.9 for the algorithm, and their §3.4.1 to understand how the pivoting is performed.)

We use `nrutil`'s routine `outerprod` instead of the more cumbersome pure Fortran 90 construction:

```
spread(a(j+1:n,j),dim=2,ncopies=n-j)*spread(a(j,j+1:n),dim=1,ncopies=n-j)
```

```
SUBROUTINE lubksb(a,indx,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Solves the set of N linear equations $A \cdot X = B$. Here the $N \times N$ matrix `a` is input, not as the original matrix A, but rather as its LU decomposition, determined by the routine `ludcmp`. `indx` is input as the permutation vector of length N returned by `ludcmp`. `b` is input as the right-hand-side vector B, also of length N, and returns with the solution vector X. `a` and `indx` are not modified by this routine and can be left in place for successive calls with different right-hand sides `b`. This routine takes into account the possibility that `b` will begin with many zero elements, so it is efficient for use in matrix inversion.

```
INTEGER(I4B) :: i,n,ii,ll
REAL(SP) :: summ
n=assert_eq(size(a,1),size(a,2),size(indx),'lubksb')
ii=0                          When ii is set to a positive value, it will become the in-
do i=1,n                          dex of the first nonvanishing element of b. We now do
    ll=indx(i)                    the forward substitution, equation (2.3.6). The only new
    summ=b(ll)                    wrinkle is to unscramble the permutation as we go.
    b(ll)=b(i)
    if (ii /= 0) then
        summ=summ-dot_product(a(i,ii:i-1),b(ii:i-1))
    else if (summ /= 0.0) then
        ii=i                  A nonzero element was encountered, so from now on we will
    end if                        have to do the dot product above.
    b(i)=summ
end do
do i=n,1,-1                   Now we do the backsubstitution, equation (2.3.7).
    b(i) = (b(i)-dot_product(a(i,i+1:n),b(i+1:n)))/a(i,i)
end do
END SUBROUTINE lubksb
```

Conceptually, the search for the first nonvanishing element of b (index `ii`) should be moved out of the first do-loop. However, in practice, the need to unscramble the permutation, and also considerations of performance

on scalar machines, cause us to write this very scalar-looking code. The performance penalty on parallel machines should be minimal.

⋆ ⋆ ⋆

Serial and parallel algorithms for tridiagonal problems are quite different. We therefore provide separate routines `tridag_ser` and `tridag_par`. In the `MODULE nr` interface file, one or the other of these (your choice) is given the generic name `tridag`. Of course, *either* version will work correctly on any computer; it is only a question of efficiency. See §22.2 for the numbering of the equation coefficients, and for a description of the parallel algorithm.

```
SUBROUTINE tridag_ser(a,b,c,r,u)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u
```

Solves for a vector `u` of size N the tridiagonal linear set given by equation (2.4.1) using a serial algorithm. Input vectors `b` (diagonal elements) and `r` (right-hand sides) have size N, while `a` and `c` (off-diagonal elements) are size $N-1$.

```
REAL(SP), DIMENSION(size(b)) :: gam        One vector of workspace, gam is needed.
INTEGER(I4B) :: n,j
REAL(SP) :: bet
n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_ser')
bet=b(1)
if (bet == 0.0) call nrerror('tridag_ser: Error at code stage 1')
```

If this happens then you should rewrite your equations as a set of order $N-1$, with u_2 trivially eliminated.

```
u(1)=r(1)/bet
do j=2,n                                   Decomposition and forward substitution.
    gam(j)=c(j-1)/bet
    bet=b(j)-a(j-1)*gam(j)
    if (bet == 0.0) &                      Algorithm fails; see below routine in Vol. 1.
        call nrerror('tridag_ser: Error at code stage 2')
    u(j)=(r(j)-a(j-1)*u(j-1))/bet
end do
do j=n-1,1,-1                              Backsubstitution.
    u(j)=u(j)-gam(j+1)*u(j+1)
end do
END SUBROUTINE tridag_ser
```

```
RECURSIVE SUBROUTINE tridag_par(a,b,c,r,u)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : tridag_ser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u
```

Solves for a vector `u` of size N the tridiagonal linear set given by equation (2.4.1) using a parallel algorithm. Input vectors `b` (diagonal elements) and `r` (right-hand sides) have size N, while `a` and `c` (off-diagonal elements) are size $N-1$.

```
INTEGER(I4B), PARAMETER :: NPAR_TRIDAG=4   Determines when serial algorithm is in-
INTEGER(I4B) :: n,n2,nm,nx                     voked.
REAL(SP), DIMENSION(size(b)/2) :: y,q,piva
REAL(SP), DIMENSION(size(b)/2-1) :: x,z
REAL(SP), DIMENSION(size(a)/2) :: pivc
n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_par')
if (n < NPAR_TRIDAG) then
    call tridag_ser(a,b,c,r,u)
else
    if (maxval(abs(b(1:n))) == 0.0) &      Algorithm fails; see below routine in Vol. 1.
```

```
        call nrerror('tridag_par: possible singular matrix')
    n2=size(y)
    nm=size(pivc)
    nx=size(x)
    piva = a(1:n-1:2)/b(1:n-1:2)              Zero the odd a's and even c's, giving x,
    pivc = c(2:n-1:2)/b(3:n:2)                   y, z, q.
    y(1:nm) = b(2:n-1:2)-piva(1:nm)*c(1:n-2:2)-pivc*a(2:n-1:2)
    q(1:nm) = r(2:n-1:2)-piva(1:nm)*r(1:n-2:2)-pivc*r(3:n:2)
    if (nm < n2) then
        y(n2) = b(n)-piva(n2)*c(n-1)
        q(n2) = r(n)-piva(n2)*r(n-1)
    end if
    x = -piva(2:n2)*a(2:n-2:2)
    z = -pivc(1:nx)*c(3:n-1:2)
    call tridag_par(x,y,z,q,u(2:n:2))         Recurse and get even u's.
    u(1) = (r(1)-c(1)*u(2))/b(1)              Substitute and get odd u's.
    u(3:n-1:2) = (r(3:n-1:2)-a(2:n-2:2)*u(2:n-2:2) &
        -c(3:n-1:2)*u(4:n:2))/b(3:n-1:2)
    if (nm == n2) u(n)=(r(n)-a(n-1)*u(n-1))/b(n)
end if
END SUBROUTINE tridag_par
```

f90 The serial version `tridag_ser` is called when the routine has recursed its way down to sufficiently small subproblems. The point at which this occurs is determined by the parameter `NPAR_TRIDAG` whose optimal value is likely machine-dependent. Notice that `tridag_ser` must here be called by its specific name, not by the generic `tridag` (which might itself be overloaded with either `tridag_ser` or `tridag_par`).

⋆ ⋆ ⋆

```
SUBROUTINE banmul(a,m1,m2,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,arth
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(OUT) :: b
    Matrix multiply b = A · x, where A is band diagonal with m1 rows below the diagonal and
    m2 rows above. If the input vector x and output vector b are of length N, then the array
    a(1:N,1:m1+m2+1) stores A as follows: The diagonal elements are in a(1:N,m1+1).
    Subdiagonal elements are in a(j:N,1:m1) (with j > 1 appropriate to the number of
    elements on each subdiagonal). Superdiagonal elements are in a(1:j,m1+2:m1+m2+1)
    with j < N appropriate to the number of elements on each superdiagonal.
INTEGER(I4B) :: m,n
n=assert_eq(size(a,1),size(b),size(x),'banmul: n')
m=assert_eq(size(a,2),m1+m2+1,'banmul: m')
b=sum(a*eoshift(spread(x,dim=2,ncopies=m), &
    dim=1,shift=arth(-m1,1,m)),dim=2)
END SUBROUTINE banmul
```

f90
```
b=sum(a*eoshift(spread(x,dim=2,ncopies=m), &
    dim=1,shift=arth(-m1,1,m)),dim=2)
```
This is a good example of Fortran 90 at both its best and its worst: best, because it allows quite subtle combinations of fully parallel operations to be built up; worst, because the resulting code is virtually incomprehensible!

What is going on becomes clearer if we imagine a temporary array y with a declaration like `REAL(SP), DIMENSION(size(a,1),size(a,2)) :: y`. Then, the above single line decomposes into

```
y=spread(x,dim=2,ncopies=m)                [Duplicate x into columns of y.]
y=eoshift(y,dim=1,shift=arth(-m1,1,m)) [Shift columns by a linear progression.]
b=sum(a*y,dim=2)                           [Multiply by the band-diagonal elements,
                                               and sum.]
```

We use here a relatively rare subcase of the `eoshift` intrinsic, using a vector value for the `shift` argument to accomplish the simultaneous shifting of a bunch of columns, by different amounts (here specified by the linear progression returned by `arth`).

If you still don't see how this accomplishes the multiplication of a band diagonal matrix by a vector, work through a simple example by hand.

```
SUBROUTINE bandec(a,m1,m2,al,indx,d)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,swap,arth
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: al
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
REAL(SP), INTENT(OUT) :: d
REAL(SP), PARAMETER :: TINY=1.0e-20_sp
```

Given an $N \times N$ band diagonal matrix **A** with m1 subdiagonal rows and m2 superdiagonal rows, compactly stored in the array a(1:N,1:m1+m2+1) as described in the comment for routine banmul, this routine constructs an LU decomposition of a rowwise permutation of **A**. The upper triangular matrix replaces a, while the lower triangular matrix is returned in al(1:N,1:m1). indx is an output vector of length N that records the row permutation effected by the partial pivoting; d is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with banbks to solve band-diagonal sets of equations.

```
INTEGER(I4B) :: i,k,l,mdum,mm,n
REAL(SP) :: dum
n=assert_eq(size(a,1),size(al,1),size(indx),'bandec: n')
mm=assert_eq(size(a,2),m1+m2+1,'bandec: mm')
mdum=assert_eq(size(al,2),m1,'bandec: mdum')
a(1:m1,:)=eoshift(a(1:m1,:),dim=2,shift=arth(m1,-1,m1))   Rearrange the storage a
d=1.0                                                          bit.
do k=1,n                                  For each row...
    l=min(m1+k,n)
    i=imaxloc(abs(a(k:l,1)))+k-1          Find the pivot element.
    dum=a(i,1)
    if (dum == 0.0) a(k,1)=TINY
```

Matrix is algorithmically singular, but proceed anyway with TINY pivot (desirable in some applications).

```
    indx(k)=i
    if (i /= k) then                      Interchange rows.
        d=-d
        call swap(a(k,1:mm),a(i,1:mm))
    end if
    do i=k+1,l                            Do the elimination.
        dum=a(i,1)/a(k,1)
        al(k,i-k)=dum
        a(i,1:mm-1)=a(i,2:mm)-dum*a(k,2:mm)
        a(i,mm)=0.0
    end do
end do
END SUBROUTINE bandec
```

`a(1:m1,:)=eoshift(a(1:m1,:),...` See similar discussion of `eoshift` for `banmul`, just above.

`i=imaxloc(abs(a(k:l,1)))+k-1` See discussion of `imaxloc` on p. 1017.

Notice that the above is *not* well parallelized for MMP machines: the outer do-loop is done N times, where N, the diagonal length, is potentially the largest dimension in the problem. Small-scale parallel (SSP) machines, and scalar machines, are not disadvantaged, because the parallelism of order `mm=m1+m2+1` in the inner loops can be enough to saturate their concurrency.

We don't know of an N-parallel algorithm for decomposing band diagonal matrices, at least one that has any reasonably concise expression in Fortran 90. Conceptually, one can view a band diagonal matrix as a *block tridiagonal* matrix, and then apply the same recursive strategy as was used in `tridag_par`. However, the implementation details of this are daunting. (We would welcome a user-contributed routine, clear, concise, and with parallelism of order N.)

```
SUBROUTINE banbks(a,m1,m2,al,indx,b)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,al
INTEGER(I4B), INTENT(IN) :: m1,m2
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
    Given the arrays a, al, and indx as returned from bandec, and given a right-hand-side
    vector b, solves the band diagonal linear equations A·x = b. The solution vector x overwrites
    b. The other input arrays are not modified, and can be left in place for successive calls with
    different right-hand sides.
INTEGER(I4B) :: i,k,l,mdum,mm,n
n=assert_eq(size(a,1),size(al,1),size(b),size(indx),'banbks: n')
mm=assert_eq(size(a,2),m1+m2+1,'banbks: mm')
mdum=assert_eq(size(al,2),m1,'banbks: mdum')
do k=1,n                        Forward substitution, unscrambling the permuted rows as we
    l=min(n,m1+k)                   go.
    i=indx(k)
    if (i /= k) call swap(b(i),b(k))
    b(k+1:l)=b(k+1:l)-al(k,1:l-k)*b(k)
end do
do i=n,1,-1                     Backsubstitution.
    l=min(mm,n-i+1)
    b(i)=(b(i)-dot_product(a(i,2:l),b(1+i:i+l-1)))/a(i,1)
end do
END SUBROUTINE banbks
```

As for `bandec`, the routine `banbks` is not parallelized on the large dimension N, though it does give the compiler the opportunity for ample small-scale parallelization inside the loops.

⋆ ⋆ ⋆

```
SUBROUTINE mprove(a,alud,indx,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : lubksb
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,alud
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
REAL(SP), DIMENSION(:), INTENT(IN) :: b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
```

Improves a solution vector `x` of the linear set of equations $A \cdot X = B$. The $N \times N$ matrix `a` and the N-dimensional vectors `b` and `x` are input. Also input is `alud`, the LU decomposition of `a` as returned by `ludcmp`, and the N-dimensional vector `indx` also returned by that routine. On output, only `x` is modified, to an improved set of values.

```
INTEGER(I4B) :: ndum
REAL(SP), DIMENSION(size(a,1)) :: r
ndum=assert_eq((/size(a,1),size(a,2),size(alud,1),size(alud,2),size(b),&
    size(x),size(indx)/),'mprove')
r=matmul(real(a,dp),real(x,dp))-real(b,dp)
```

Calculate the right-hand side, accumulating the residual in double precision.

```
call lubksb(alud,indx,r)        Solve for the error term,
x=x-r                           and subtract it from the old solution.
END SUBROUTINE mprove
```

`assert_eq((/.../),'mprove')` This overloaded version of the `nrutil` routine `assert_eq` makes use of a trick for passing a variable number of scalar arguments to a routine: Put them into an array constructor, `(/.../)`, and pass the array. The receiving routine can use the `size` intrinsic to count them. The technique has some obvious limitations: All the arguments in the array must be of the same type; and the arguments are passed, in effect, by *value*, not by address, so they must be, in effect, `INTENT(IN)`.

`r=matmul(real(a,dp),real(x,dp))-real(b,dp)` Since Fortran 90's elemental intrinsics operate with the type of their arguments, we can use the `real(...,dp)`'s to force the `matmul` matrix multiplication to be done in double precision, which is what we want. In Fortran 77, we would have to do the matrix multiplication with temporary double precision variables, both inconvenient and (since Fortran 77 has no dynamic memory allocation) a waste of memory.

⋆ ⋆ ⋆

```
SUBROUTINE svbksb_sp(u,w,v,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
REAL(SP), DIMENSION(:,:), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:), INTENT(IN) :: w,b
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
```

Solves $A \cdot X = B$ for a vector X, where A is specified by the arrays `u`, `v`, `w` as returned by `svdcmp`. Here `u` is $M \times N$, `v` is $N \times N$, and `w` is of length N. `b` is the M-dimensional input right-hand side. `x` is the N-dimensional output solution vector. No input quantities are destroyed, so the routine may be called sequentially with different `b`'s.

```
INTEGER(I4B) :: mdum,ndum
REAL(SP), DIMENSION(size(x)) :: tmp
mdum=assert_eq(size(u,1),size(b),'svbksb_sp: mdum')
ndum=assert_eq((/size(u,2),size(v,1),size(v,2),size(w),size(x)/),&
    'svbksb_sp: ndum')
where (w /= 0.0)
    tmp=matmul(b,u)/w           Calculate diag(1/w_j) U^T B,
elsewhere
    tmp=0.0                     but replace 1/w_j by zero if w_j = 0.
end where
```

```
x=matmul(v,tmp)                    Matrix multiply by V to get answer.
END SUBROUTINE svbksb_sp
```

f90 `where (w /= 0.0)...tmp=...elsewhere...tmp=` Normally, when a `where ...elsewhere` construction is used to set a variable (here `tmp`) to one or another value, we like to replace it with a `merge` expression. Here, however, the `where` is required to guarantee that a division by zero doesn't occur. The rule is that `where` will *never* evaluate expressions that are excluded by the mask in the `where` line, but other constructions, like `merge`, *might* perform speculative evaluation of more than one possible outcome before selecting the applicable one.

Because singular value decomposition is something that one often wants to do in double precision, we include a double-precision version. In `nr`, the single- and double-precision versions are overloaded onto the name `svbksb`.

```
SUBROUTINE svbksb_dp(u,w,v,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,v
REAL(DP), DIMENSION(:), INTENT(IN) :: w,b
REAL(DP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B) :: mdum,ndum
REAL(DP), DIMENSION(size(x)) :: tmp
mdum=assert_eq(size(u,1),size(b),'svbksb_dp: mdum')
ndum=assert_eq((/size(u,2),size(v,1),size(v,2),size(w),size(x)/),&
    'svbksb_dp: ndum')
where (w /= 0.0)
    tmp=matmul(b,u)/w
elsewhere
    tmp=0.0
end where
x=matmul(v,tmp)
END SUBROUTINE svbksb_dp
```

```
SUBROUTINE svdcmp_sp(a,w,v)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerprod
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: w
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
    Given an M × N matrix a, this routine computes its singular value decomposition, A =
    U · W · V^T. The matrix U replaces a on output. The diagonal matrix of singular values
    W is output as the N-dimensional vector w. The N × N matrix V (not the transpose V^T)
    is output as v.
INTEGER(I4B) :: i,its,j,k,l,m,n,nm
REAL(SP) :: anorm,c,f,g,h,s,scale,x,y,z
REAL(SP), DIMENSION(size(a,1)) :: tempm
REAL(SP), DIMENSION(size(a,2)) :: rv1,tempn
m=size(a,1)
n=assert_eq(size(a,2),size(v,1),size(v,2),size(w),'svdcmp_sp')
g=0.0
scale=0.0
do i=1,n                           Householder reduction to bidiagonal form.
    l=i+1
    rv1(i)=scale*g
    g=0.0
    scale=0.0
    if (i <= m) then
```

```
            scale=sum(abs(a(i:m,i)))
            if (scale /= 0.0) then
                a(i:m,i)=a(i:m,i)/scale
                s=dot_product(a(i:m,i),a(i:m,i))
                f=a(i,i)
                g=-sign(sqrt(s),f)
                h=f*g-s
                a(i,i)=f-g
                tempn(l:n)=matmul(a(i:m,i),a(i:m,l:n))/h
                a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
                a(i:m,i)=scale*a(i:m,i)
            end if
        end if
        w(i)=scale*g
        g=0.0
        scale=0.0
        if ((i <= m) .and. (i /= n)) then
            scale=sum(abs(a(i,l:n)))
            if (scale /= 0.0) then
                a(i,l:n)=a(i,l:n)/scale
                s=dot_product(a(i,l:n),a(i,l:n))
                f=a(i,l)
                g=-sign(sqrt(s),f)
                h=f*g-s
                a(i,l)=f-g
                rv1(l:n)=a(i,l:n)/h
                tempm(l:m)=matmul(a(l:m,l:n),a(i,l:n))
                a(l:m,l:n)=a(l:m,l:n)+outerprod(tempm(l:m),rv1(l:n))
                a(i,l:n)=scale*a(i,l:n)
            end if
        end if
    end do
    anorm=maxval(abs(w)+abs(rv1))
    do i=n,1,-1                                  Accumulation of right-hand transformations.
        if (i < n) then
            if (g /= 0.0) then
                v(l:n,i)=(a(i,l:n)/a(i,l))/g      Double division to avoid possible under-
                tempn(l:n)=matmul(a(i,l:n),v(l:n,l:n))      flow.
                v(l:n,l:n)=v(l:n,l:n)+outerprod(v(l:n,i),tempn(l:n))
            end if
            v(i,l:n)=0.0
            v(l:n,i)=0.0
        end if
        v(i,i)=1.0
        g=rv1(i)
        l=i
    end do
    do i=min(m,n),1,-1                           Accumulation of left-hand transformations.
        l=i+1
        g=w(i)
        a(i,l:n)=0.0
        if (g /= 0.0) then
            g=1.0_sp/g
            tempn(l:n)=(matmul(a(l:m,i),a(l:m,l:n))/a(i,i))*g
            a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
            a(i:m,i)=a(i:m,i)*g
        else
            a(i:m,i)=0.0
        end if
        a(i,i)=a(i,i)+1.0_sp
    end do
    do k=n,1,-1                                  Diagonalization of the bidiagonal form: Loop over
        do its=1,30                                  singular values, and over allowed iterations.
            do l=k,1,-1                          Test for splitting.
```

```
    nm=l-1
    if ((abs(rv1(l))+anorm) == anorm) exit
      Note that rv1(1) is always zero, so can never fall through bottom of loop.
    if ((abs(w(nm))+anorm) == anorm) then
        c=0.0                       Cancellation of rv1(l), if l > 1.
        s=1.0
        do i=l,k
            f=s*rv1(i)
            rv1(i)=c*rv1(i)
            if ((abs(f)+anorm) == anorm) exit
            g=w(i)
            h=pythag(f,g)
            w(i)=h
            h=1.0_sp/h
            c= (g*h)
            s=-(f*h)
            tempm(1:m)=a(1:m,nm)
            a(1:m,nm)=a(1:m,nm)*c+a(1:m,i)*s
            a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
        end do
        exit
    end if
end do
z=w(k)
if (l == k) then                    Convergence.
    if (z < 0.0) then               Singular value is made nonnegative.
        w(k)=-z
        v(1:n,k)=-v(1:n,k)
    end if
    exit
end if
if (its == 30) call nrerror('svdcmp_sp: no convergence in svdcmp')
x=w(l)                              Shift from bottom 2-by-2 minor.
nm=k-1
y=w(nm)
g=rv1(nm)
h=rv1(k)
f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0_sp*h*y)
g=pythag(f,1.0_sp)
f=((x-z)*(x+z)+h*((y/(f+sign(g,f)))-h))/x
c=1.0                               Next QR transformation:
s=1.0
do j=l,nm
    i=j+1
    g=rv1(i)
    y=w(i)
    h=s*g
    g=c*g
    z=pythag(f,h)
    rv1(j)=z
    c=f/z
    s=h/z
    f= (x*c)+(g*s)
    g=-(x*s)+(g*c)
    h=y*s
    y=y*c
    tempn(1:n)=v(1:n,j)
    v(1:n,j)=v(1:n,j)*c+v(1:n,i)*s
    v(1:n,i)=-tempn(1:n)*s+v(1:n,i)*c
    z=pythag(f,h)
    w(j)=z                          Rotation can be arbitrary if z = 0.
    if (z /= 0.0) then
        z=1.0_sp/z
        c=f*z
```

```
                s=h*z
            end if
            f= (c*g)+(s*y)
            x=-(s*g)+(c*y)
            tempm(1:m)=a(1:m,j)
            a(1:m,j)=a(1:m,j)*c+a(1:m,i)*s
            a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
        end do
        rv1(l)=0.0
        rv1(k)=f
        w(k)=x
    end do
end do
END SUBROUTINE svdcmp_sp
```

The SVD algorithm implemented above does not parallelize very well. There are two parts to the algorithm. The first, reduction to bidiagonal form, can be parallelized. The second, the iterative diagonalization of the bidiagonal form, uses QR transformations that are intrinsically serial. There have been proposals for parallel SVD algorithms [2], but we do not have sufficient experience with them yet to recommend them over the well-established serial algorithm.

`tempn(l:n)=matmul...a(i:m,l:n)=...outerprod...` Here is an example of an update as in equation (22.1.6). In this case b_i is independent of i: It is simply `1/h`. The lines beginning `tempm(l:m)=matmul` about 16 lines down are of a similar form, but with the terms in the opposite order in the `matmul`.

As with `svbksb`, single- and double-precision versions of the routines are overloaded onto the name `svdcmp` in `nr`.

```
SUBROUTINE svdcmp_dp(a,w,v)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,outerprod
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(DP), DIMENSION(:), INTENT(OUT) :: w
REAL(DP), DIMENSION(:,:), INTENT(OUT) :: v
INTEGER(I4B) :: i,its,j,k,l,m,n,nm
REAL(DP) :: anorm,c,f,g,h,s,scale,x,y,z
REAL(DP), DIMENSION(size(a,1)) :: tempm
REAL(DP), DIMENSION(size(a,2)) :: rv1,tempn
m=size(a,1)
n=assert_eq(size(a,2),size(v,1),size(v,2),size(w),'svdcmp_dp')
g=0.0
scale=0.0
do i=1,n
    l=i+1
    rv1(i)=scale*g
    g=0.0
    scale=0.0
    if (i <= m) then
        scale=sum(abs(a(i:m,i)))
        if (scale /= 0.0) then
            a(i:m,i)=a(i:m,i)/scale
            s=dot_product(a(i:m,i),a(i:m,i))
            f=a(i,i)
            g=-sign(sqrt(s),f)
```

```
            h=f*g-s
            a(i,i)=f-g
            tempn(l:n)=matmul(a(i:m,i),a(i:m,l:n))/h
            a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
            a(i:m,i)=scale*a(i:m,i)
        end if
    end if
    w(i)=scale*g
    g=0.0
    scale=0.0
    if ((i <= m) .and. (i /= n)) then
        scale=sum(abs(a(i,l:n)))
        if (scale /= 0.0) then
            a(i,l:n)=a(i,l:n)/scale
            s=dot_product(a(i,l:n),a(i,l:n))
            f=a(i,l)
            g=-sign(sqrt(s),f)
            h=f*g-s
            a(i,l)=f-g
            rv1(l:n)=a(i,l:n)/h
            tempm(l:m)=matmul(a(l:m,l:n),a(i,l:n))
            a(l:m,l:n)=a(l:m,l:n)+outerprod(tempm(l:m),rv1(l:n))
            a(i,l:n)=scale*a(i,l:n)
        end if
    end if
end do
anorm=maxval(abs(w)+abs(rv1))
do i=n,1,-1
    if (i < n) then
        if (g /= 0.0) then
            v(l:n,i)=(a(i,l:n)/a(i,l))/g
            tempn(l:n)=matmul(a(i,l:n),v(l:n,l:n))
            v(l:n,l:n)=v(l:n,l:n)+outerprod(v(l:n,i),tempn(l:n))
        end if
        v(i,l:n)=0.0
        v(l:n,i)=0.0
    end if
    v(i,i)=1.0
    g=rv1(i)
    l=i
end do
do i=min(m,n),1,-1
    l=i+1
    g=w(i)
    a(i,l:n)=0.0
    if (g /= 0.0) then
        g=1.0_dp/g
        tempn(l:n)=(matmul(a(l:m,i),a(l:m,l:n))/a(i,i))*g
        a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),tempn(l:n))
        a(i:m,i)=a(i:m,i)*g
    else
        a(i:m,i)=0.0
    end if
    a(i,i)=a(i,i)+1.0_dp
end do
do k=n,1,-1
    do its=1,30
        do l=k,1,-1
            nm=l-1
            if ((abs(rv1(l))+anorm) == anorm) exit
            if ((abs(w(nm))+anorm) == anorm) then
                c=0.0
                s=1.0
                do i=l,k
```

```
            f=s*rv1(i)
            rv1(i)=c*rv1(i)
            if ((abs(f)+anorm) == anorm) exit
            g=w(i)
            h=pythag(f,g)
            w(i)=h
            h=1.0_dp/h
            c= (g*h)
            s=-(f*h)
            tempm(1:m)=a(1:m,nm)
            a(1:m,nm)=a(1:m,nm)*c+a(1:m,i)*s
            a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
        end do
        exit
    end if
end do
z=w(k)
if (l == k) then
    if (z < 0.0) then
        w(k)=-z
        v(1:n,k)=-v(1:n,k)
    end if
    exit
end if
if (its == 30) call nrerror('svdcmp_dp: no convergence in svdcmp')
x=w(l)
nm=k-1
y=w(nm)
g=rv1(nm)
h=rv1(k)
f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0_dp*h*y)
g=pythag(f,1.0_dp)
f=((x-z)*(x+z)+h*((y/(f+sign(g,f)))-h))/x
c=1.0
s=1.0
do j=l,nm
    i=j+1
    g=rv1(i)
    y=w(i)
    h=s*g
    g=c*g
    z=pythag(f,h)
    rv1(j)=z
    c=f/z
    s=h/z
    f= (x*c)+(g*s)
    g=-(x*s)+(g*c)
    h=y*s
    y=y*c
    tempn(1:n)=v(1:n,j)
    v(1:n,j)=v(1:n,j)*c+v(1:n,i)*s
    v(1:n,i)=-tempn(1:n)*s+v(1:n,i)*c
    z=pythag(f,h)
    w(j)=z
    if (z /= 0.0) then
        z=1.0_dp/z
        c=f*z
        s=h*z
    end if
    f= (c*g)+(s*y)
    x=-(s*g)+(c*y)
    tempm(1:m)=a(1:m,j)
    a(1:m,j)=a(1:m,j)*c+a(1:m,i)*s
    a(1:m,i)=-tempm(1:m)*s+a(1:m,i)*c
```

```
        end do
        rv1(l)=0.0
        rv1(k)=f
        w(k)=x
    end do
end do
END SUBROUTINE svdcmp_dp
```

```
FUNCTION pythag_sp(a,b)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: pythag_sp
```
Computes $(a^2 + b^2)^{1/2}$ without destructive underflow or overflow.
```
REAL(SP) :: absa,absb
absa=abs(a)
absb=abs(b)
if (absa > absb) then
    pythag_sp=absa*sqrt(1.0_sp+(absb/absa)**2)
else
    if (absb == 0.0) then
        pythag_sp=0.0
    else
        pythag_sp=absb*sqrt(1.0_sp+(absa/absb)**2)
    end if
end if
END FUNCTION pythag_sp
```

```
FUNCTION pythag_dp(a,b)
USE nrtype
IMPLICIT NONE
REAL(DP), INTENT(IN) :: a,b
REAL(DP) :: pythag_dp
REAL(DP) :: absa,absb
absa=abs(a)
absb=abs(b)
if (absa > absb) then
    pythag_dp=absa*sqrt(1.0_dp+(absb/absa)**2)
else
    if (absb == 0.0) then
        pythag_dp=0.0
    else
        pythag_dp=absb*sqrt(1.0_dp+(absa/absb)**2)
    end if
end if
END FUNCTION pythag_dp
```

⋆ ⋆ ⋆

```
SUBROUTINE cyclic(a,b,c,alpha,beta,r,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : tridag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN):: a,b,c,r
REAL(SP), INTENT(IN) :: alpha,beta
REAL(SP), DIMENSION(:), INTENT(OUT):: x
```

Solves the "cyclic" set of linear equations given by equation (2.7.9). a, b, c, and r are input vectors, while x is the output solution vector, all of the same size. alpha and beta are the corner entries in the matrix. The input is not modified.

```
INTEGER(I4B) :: n
REAL(SP) :: fact,gamma
REAL(SP), DIMENSION(size(x)) :: bb,u,z
n=assert_eq((/size(a),size(b),size(c),size(r),size(x)/),'cyclic')
call assert(n > 2, 'cyclic arg')
gamma=-b(1)                                  Avoid subtraction error in forming bb(1).
bb(1)=b(1)-gamma                             Set up the diagonal of the modified tridiag-
bb(n)=b(n)-alpha*beta/gamma                      onal system.
bb(2:n-1)=b(2:n-1)
call tridag(a(2:n),bb,c(1:n-1),r,x)          Solve A · x = r.
u(1)=gamma                                   Set up the vector u.
u(n)=alpha
u(2:n-1)=0.0
call tridag(a(2:n),bb,c(1:n-1),u,z)          Solve A · z = u.
fact=(x(1)+beta*x(n)/gamma)/(1.0_sp+z(1)+beta*z(n)/gamma)      Form v·x/(1+v·z).
x=x-fact*z                                   Now get the solution vector x.
END SUBROUTINE cyclic
```

The parallelism in cyclic is in tridag. Users with multiprocessor machines will want to be sure that, in nrutil, they have set the name tridag to be overloaded with tridag_par instead of tridag_ser.

⋆ ⋆ ⋆

The routines sprsin, sprsax, sprstx, sprstp, and sprsdiag give roughly equivalent functionality to the corresponding Fortran 77 routines, but they are *not* plug compatible. Instead, they take advantage of (and illustrate) several Fortran 90 features that are not present in Fortran 77.

In the module nrtype we define a TYPE sprs2_sp for two-dimensional sparse, square, matrices, in single precision, as follows

```
TYPE sprs2_sp
   INTEGER(I4B) :: n,len
   REAL(SP), DIMENSION(:), POINTER :: val
   INTEGER(I4B), DIMENSION(:), POINTER :: irow
   INTEGER(I4B), DIMENSION(:), POINTER :: jcol
END TYPE sprs2_sp
```

This has much less structure to it than the "row-indexed sparse storage mode" used in Volume 1. Here, a sparse matrix is just a list of values, and corresponding lists giving the row and column number that each value is in. Two integers n and len give, respectively, the underlying size (number of rows or columns) in the full matrix, and the number of stored nonzero values. While the previously used row-indexed scheme can be somewhat more efficient for serial machines, it does not parallelize conveniently, while this one does (though with some caveats; see below).

```
SUBROUTINE sprsin_sp(a,thresh,sa)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), INTENT(IN) :: thresh
TYPE(sprs2_sp), INTENT(OUT) :: sa
```

Converts a square matrix a to sparse storage format as sa. Only elements of a with magnitude $\geq$ thresh are retained.

```
INTEGER(I4B) :: n,len
LOGICAL(LGT), DIMENSION(size(a,1),size(a,2)) :: mask
n=assert_eq(size(a,1),size(a,2),'sprsin_sp')
mask=abs(a)>thresh
len=count(mask)                How many elements to store?
allocate(sa%val(len),sa%irow(len),sa%jcol(len))
sa%n=n
sa%len=len
sa%val=pack(a,mask)            Grab the values, row, and column numbers.
sa%irow=pack(spread(arth(1,1,n),2,n),mask)
sa%jcol=pack(spread(arth(1,1,n),1,n),mask)
END SUBROUTINE sprsin_sp
```

```
SUBROUTINE sprsin_dp(a,thresh,sa)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: a
REAL(DP), INTENT(IN) :: thresh
TYPE(sprs2_dp), INTENT(OUT) :: sa
INTEGER(I4B) :: n,len
LOGICAL(LGT), DIMENSION(size(a,1),size(a,2)) :: mask
n=assert_eq(size(a,1),size(a,2),'sprsin_dp')
mask=abs(a)>thresh
len=count(mask)
allocate(sa%val(len),sa%irow(len),sa%jcol(len))
sa%n=n
sa%len=len
sa%val=pack(a,mask)
sa%irow=pack(spread(arth(1,1,n),2,n),mask)
sa%jcol=pack(spread(arth(1,1,n),1,n),mask)
END SUBROUTINE sprsin_dp
```

f90 Note that the routines sprsin_sp and sprsin_dp — single and double precision versions of the same algorithm — are overloaded onto the name sprsin in module nr. We supply both forms because the routine linbcg, below, works in double precision.

`sa%irow=pack(spread(arth(1,1,n),2,n),mask)` The trick here is to use the same mask, `abs(a)>thresh`, in three consecutive `pack` expressions, thus guaranteeing that the corresponding elements of the array argument get selected for packing. The first time, we get the desired matrix element values. The second time (above code fragment), we construct a matrix with each element having the value of its *row* number. The third time, we construct a matrix with each element having the value of its *column* number.

```
SUBROUTINE sprsax_sp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
    Multiply a matrix sa in sparse matrix format by a vector x, giving a vector b.
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprsax_sp')
b=0.0_sp
call scatter_add(b,sa%val*x(sa%jcol),sa%irow)
  Each sparse matrix entry adds a term to some component of b.
END SUBROUTINE sprsax_sp


SUBROUTINE sprsax_dp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprsax_dp')
b=0.0_dp
call scatter_add(b,sa%val*x(sa%jcol),sa%irow)
END SUBROUTINE sprsax_dp
```

`call scatter_add(b,sa%val*x(sa%jcol),sa%irow)` Since more than one component of the middle vector argument will, in general, need to be added into the same component of `b`, we must resort to a call to the `nrutil` routine `scatter_add` to achieve parallelism. *However*, this parallelism is achieved only if a parallel version of `scatter_add` is available! As we have discussed previously (p. 984), Fortran 90 does not provide any scatter-with-combine (here, scatter-with-add) facility, insisting instead that indexed operations yield non-colliding addresses. Luckily, almost all parallel machines do provide such a facility as a library program. In HPF, for example, the equivalent of `scatter_add` is `SUM_SCATTER`.

The call to `scatter_add` above is equivalent to the do-loop

```
b=0.0
do k=1,sa%len
   b(sa%irow(k))=b(sa%irow(k))+sa%val(k)*x(sa%jcol(k))
end do
```

```
SUBROUTINE sprstx_sp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION (:), INTENT(IN) :: x
REAL(SP), DIMENSION (:), INTENT(OUT) :: b
    Multiply the transpose of a matrix sa in sparse matrix format by a vector x, giving a vector b.
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprstx_sp')
b=0.0_sp
call scatter_add(b,sa%val*x(sa%irow),sa%jcol)
  Each sparse matrix entry adds a term to some component of b.
END SUBROUTINE sprstx_sp
```

```
SUBROUTINE sprstx_dp(sa,x,b)
USE nrtype; USE nrutil, ONLY : assert_eq,scatter_add
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION (:), INTENT(IN) :: x
REAL(DP), DIMENSION (:), INTENT(OUT) :: b
INTEGER(I4B) :: ndum
ndum=assert_eq(sa%n,size(x),size(b),'sprstx_dp')
b=0.0_dp
call scatter_add(b,sa%val*x(sa%irow),sa%jcol)
END SUBROUTINE sprstx_dp
```

Precisely the same comments as for sprsax apply to sprstx. The call to scatter_add is here equivalent to

```
b=0.0
do k=1,sa%len
   b(sa%jcol(k))=b(sa%jcol(k))+sa%val(k)*x(sa%irow(k))
end do
```

```
SUBROUTINE sprstp(sa)
USE nrtype
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(INOUT) :: sa
   Replaces sa, in sparse matrix format, by its transpose.
INTEGER(I4B), DIMENSION(:), POINTER :: temp
temp=>sa%irow                We need only swap the row and column pointers.
sa%irow=>sa%jcol
sa%jcol=>temp
END SUBROUTINE sprstp
```

```
SUBROUTINE sprsdiag_sp(sa,b)
USE nrtype; USE nrutil, ONLY : array_copy,assert_eq
IMPLICIT NONE
TYPE(sprs2_sp), INTENT(IN) :: sa
REAL(SP), DIMENSION(:), INTENT(OUT) :: b
   Extracts the diagonal of a matrix sa in sparse matrix format into a vector b.
REAL(SP), DIMENSION(size(b)) :: val
INTEGER(I4B) :: k,l,ndum,nerr
INTEGER(I4B), DIMENSION(size(b)) :: i
LOGICAL(LGT), DIMENSION(:), ALLOCATABLE :: mask
ndum=assert_eq(sa%n,size(b),'sprsdiag_sp')
l=sa%len
allocate(mask(l))
mask = (sa%irow(1:l) == sa%jcol(1:l))      Find diagonal elements.
call array_copy(pack(sa%val(1:l),mask),val,k,nerr)      Grab the values...
i(1:k)=pack(sa%irow(1:l),mask)             ...and their locations.
deallocate(mask)
b=0.0                                      Zero b because zero values not stored in sa.
b(i(1:k))=val(1:k)                         Scatter values into correct slots.
END SUBROUTINE sprsdiag_sp
```

```
SUBROUTINE sprsdiag_dp(sa,b)
USE nrtype; USE nrutil, ONLY : array_copy,assert_eq
IMPLICIT NONE
TYPE(sprs2_dp), INTENT(IN) :: sa
REAL(DP), DIMENSION(:), INTENT(OUT) :: b
REAL(DP), DIMENSION(size(b)) :: val
INTEGER(I4B) :: k,l,ndum,nerr
INTEGER(I4B), DIMENSION(size(b)) :: i
LOGICAL(LGT), DIMENSION(:), ALLOCATABLE :: mask
ndum=assert_eq(sa%n,size(b),'sprsdiag_dp')
l=sa%len
allocate(mask(l))
mask = (sa%irow(1:l) == sa%jcol(1:l))
call array_copy(pack(sa%val(1:l),mask),val,k,nerr)
i(1:k)=pack(sa%irow(1:l),mask)
deallocate(mask)
b=0.0
b(i(1:k))=val(1:k)
END SUBROUTINE sprsdiag_dp
```

f90 `call array_copy(pack(sa%val(1:l),mask),val,k,nerr)` We use the `nrutil` routine `array_copy` because we don't know in advance how many nonzero diagonal elements will be selected by `mask`. Of course we could count them with a `count(mask)`, but this is an extra step, and inefficient on scalar machines.

`i(1:k)=pack(sa%irow(1:l),mask)` Using the same mask, we pick out the corresponding locations of the diagonal elements. No need to use `array_copy` now, since we know the value of `k`.

`b(i(1:k))=val(1:k)` Finally, we can put each element in the right place. Notice that if the sparse matrix is ill-formed, with more than one value stored for the same diagonal element (which should not happen!) then the vector subscript `i(1:k)` is a "many-one section" and its use on the left-hand side is illegal.

⋆ ⋆ ⋆

```
SUBROUTINE linbcg(b,x,itol,tol,itmax,iter,err)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : atimes,asolve,snrm
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: b
REAL(DP), DIMENSION(:), INTENT(INOUT) :: x
INTEGER(I4B), INTENT(IN) :: itol,itmax
REAL(DP), INTENT(IN) :: tol
INTEGER(I4B), INTENT(OUT) :: iter
REAL(DP), INTENT(OUT) :: err
REAL(DP), PARAMETER :: EPS=1.0e-14_dp
```

Double precision is a good idea in this routine.

Solves $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for `x`, given `b` of the same length, by the iterative biconjugate gradient method. On input `x` should be set to an initial guess of the solution (or all zeros); `itol` is 1,2,3, or 4, specifying which convergence test is applied (see text); `itmax` is the maximum number of allowed iterations; and `tol` is the desired convergence tolerance. On output, `x` is reset to the improved solution, `iter` is the number of iterations actually taken, and `err` is the estimated error. The matrix $\mathbf{A}$ is referenced only through the user-supplied routines `atimes`, which computes the product of either $\mathbf{A}$ or its transpose on a vector; and `asolve`,

which solves $\tilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$ or $\tilde{\mathbf{A}}^T \cdot \mathbf{x} = \mathbf{b}$ for some preconditioner matrix $\tilde{\mathbf{A}}$ (possibly the trivial diagonal part of $\mathbf{A}$).

```
INTEGER(I4B) :: n
REAL(DP) :: ak,akden,bk,bkden,bknum,bnrm,dxnrm,xnrm,zm1nrm,znrm
REAL(DP), DIMENSION(size(b)) :: p,pp,r,rr,z,zz
n=assert_eq(size(b),size(x),'linbcg')
iter=0
call atimes(x,r,0)                          Calculate initial residual. Input to atimes is
r=b-r                                           x(1:n), output is r(1:n); the final 0
rr=r                                            indicates that the matrix (not its trans-
!call atimes(r,rr,0)                            pose) is to be used.
  Uncomment this line to get the "minimum residual" variant of the algorithm.
select case(itol)                           Calculate norms for use in stopping criterion,
    case(1)                                     and initialize z.
        bnrm=snrm(b,itol)
        call asolve(r,z,0)                  Input to asolve is r(1:n), output is z(1:n);
    case(2)                                     the final 0 indicates that the matrix Ã
        call asolve(b,z,0)                      (not its transpose) is to be used.
        bnrm=snrm(z,itol)
        call asolve(r,z,0)
    case(3:4)
        call asolve(b,z,0)
        bnrm=snrm(z,itol)
        call asolve(r,z,0)
        znrm=snrm(z,itol)
    case default
        call nrerror('illegal itol in linbcg')
end select
do                                          Main loop.
    if (iter > itmax) exit
    iter=iter+1
    call asolve(rr,zz,1)                    Final 1 indicates use of transpose matrix Ã^T.
    bknum=dot_product(z,rr)                 Calculate coefficient bk and direction vectors
    if (iter == 1) then                         p and pp.
        p=z
        pp=zz
    else
        bk=bknum/bkden
        p=bk*p+z
        pp=bk*pp+zz
    end if
    bkden=bknum                             Calculate coefficient ak, new iterate x, and
    call atimes(p,z,0)                          new residuals r and rr.
    akden=dot_product(z,pp)
    ak=bknum/akden
    call atimes(pp,zz,1)
    x=x+ak*p
    r=r-ak*z
    rr=rr-ak*zz
    call asolve(r,z,0)                      Solve Ã·z = r and check stopping criterion.
    select case(itol)
        case(1)
            err=snrm(r,itol)/bnrm
        case(2)
            err=snrm(z,itol)/bnrm
        case(3:4)
            zm1nrm=znrm
            znrm=snrm(z,itol)
            if (abs(zm1nrm-znrm) > EPS*znrm) then
                dxnrm=abs(ak)*snrm(p,itol)
                err=znrm/abs(zm1nrm-znrm)*dxnrm
            else
                err=znrm/bnrm               Error may not be accurate, so loop again.
                cycle
```

```
            end if
            xnrm=snrm(x,itol)
            if (err <= 0.5_dp*xnrm) then
                err=err/xnrm
            else
                err=znrm/bnrm                Error may not be accurate, so loop again.
                cycle
            end if
        end select
        write (*,*) ' iter=',iter,' err=',err
        if (err <= tol) exit
    end do
    END SUBROUTINE linbcg
```

`case default...call nrerror('illegal itol in linbcg')` It's *always* a good idea to trap errors when the value of a case construction is supplied externally to the routine, as here.

```
FUNCTION snrm(sx,itol)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: sx
INTEGER(I4B), INTENT(IN) :: itol
REAL(DP) :: snrm
    Compute one of two norms for a vector sx, as signaled by itol. Used by linbcg.
if (itol <= 3) then
    snrm=sqrt(dot_product(sx,sx))            Vector magnitude norm.
else
    snrm=maxval(abs(sx))                     Largest component norm.
end if
END FUNCTION snrm
```

```
SUBROUTINE atimes(x,r,itrnsp)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : sprsax,sprstx        DOUBLE PRECISION versions of sprsax and sprstx.
USE xlinbcg_data                    The matrix is accessed through this module.
REAL(DP), DIMENSION(:), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(OUT) :: r
INTEGER(I4B), INTENT(IN) :: itrnsp
INTEGER(I4B) :: n
n=assert_eq(size(x),size(r),'atimes')
if (itrnsp == 0) then
    call sprsax(sa,x,r)
else
    call sprstx(sa,x,r)
end if
END SUBROUTINE atimes
```

```
SUBROUTINE asolve(b,x,itrnsp)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : sprsdiag          DOUBLE PRECISION version of sprsdiag.
USE xlinbcg_data                 The matrix is accessed through this module.
REAL(DP), DIMENSION(:), INTENT(IN) :: b
REAL(DP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B), INTENT(IN) :: itrnsp
INTEGER(I4B) :: ndum
ndum=assert_eq(size(b),size(x),'asolve')
call sprsdiag(sa,x)
```

The matrix $\tilde{\mathbf{A}}$ is taken to be the diagonal part of $\mathbf{A}$. Since the transpose matrix has the same diagonal, the flag itrnsp is not used.

```
if (any(x == 0.0)) call nrerror('asolve: singular diagonal matrix')
x=b/x
END SUBROUTINE asolve
```

f90 The routines atimes and asolve are examples of user-supplied routines that interface linbcg to a user-supplied method for multiplying the user's sparse matrix by a vector, and for solving the preconditioner matrix equation. Here, we have used these routines to connect linbcg to the sparse matrix machinery developed above. If we were instead using the different sparse matrix machinery of Volume 1, we would modify atimes and asolve accordingly.

USE xlinbcg_data This user-supplied module is assumed to have sa (the sparse matrix) in it.

⋆ ⋆ ⋆

```
FUNCTION vander(x,q)
USE nrtype; USE nrutil, ONLY : assert_eq,outerdiff
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x,q
REAL(DP), DIMENSION(size(x)) :: vander
```

Solves the Vandermonde linear system $\sum_{i=1}^{N} x_i^{k-1} w_i = q_k \ (k = 1, \ldots, N)$. Input consists of the vectors x and q of length N. The solution w (also of length N) is returned in vander.

```
REAL(DP), DIMENSION(size(x)) :: c
REAL(DP), DIMENSION(size(x),size(x)) :: a
INTEGER(I4B) :: i,n
n=assert_eq(size(x),size(q),'vander')
if (n == 1) then
    vander(1)=q(1)
else
    c(:)=0.0                                  Initialize array.
    c(n)=-x(1)                                Coefficients of the master polynomial are found
    do i=2,n                                      by recursion.
        c(n+1-i:n-1)=c(n+1-i:n-1)-x(i)*c(n+2-i:n)
        c(n)=c(n)-x(i)
    end do
    a(:,:)=outerdiff(x,x)                     Make vector w_j = ∏_{n≠j}(x_j − x_n).
    vander(:)=product(a,dim=2,mask=(a /= 0.0))
```

Now do synthetic division by $x - x_j$. The division for all x_j can be done in parallel (on a parallel machine), since the : in the loop below is over j.

```
    a(:,1)=-c(1)/x(:)
    do i=2,n
        a(:,i)=-(c(i)-a(:,i-1))/x(:)
    end do
    vander(:)=matmul(a,q)/vander(:)           Solve linear system and supply denomina-
end if                                            tor.
END FUNCTION vander
```

f90 a=outerdiff...w=product... Here is an example of the coding of equation (22.1.4). Since in this case the product is over the second index (n in $x_j - x_n$), we have dim=2 in the product.

```
FUNCTION toeplz(r,y)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: r,y
REAL(SP), DIMENSION(size(y)) :: toeplz
```

Solves the Toeplitz system $\sum_{j=1}^{N} R_{(N+i-j)}x_j = y_i$ $(i = 1, \ldots, N)$. The Toeplitz matrix need not be symmetric. y (of length N) and r (of length $2N - 1$) are input arrays; the solution x (of length N) is returned in toeplz.

```
INTEGER(I4B) :: m,m1,n,ndum
REAL(SP) :: sd,sgd,sgn,shn,sxn
REAL(SP), DIMENSION(size(y)) :: g,h,t
n=size(y)
ndum=assert_eq(2*n-1,size(r),'toeplz: ndum')
if (r(n) == 0.0) call nrerror('toeplz: initial singular minor')
toeplz(1)=y(1)/r(n)                          Initialize for the recursion.
if (n == 1) RETURN
g(1)=r(n-1)/r(n)
h(1)=r(n+1)/r(n)
do m=1,n                                     Main loop over the recursion.
    m1=m+1
    sxn=-y(m1)+dot_product(r(n+1:n+m),toeplz(m:1:-1))
      Compute numerator and denominator for x,
    sd=-r(n)+dot_product(r(n+1:n+m),g(1:m))
    if (sd == 0.0) exit
    toeplz(m1)=sxn/sd                        whence x.
    toeplz(1:m)=toeplz(1:m)-toeplz(m1)*g(m:1:-1)
    if (m1 == n) RETURN
    sgn=-r(n-m1)+dot_product(r(n-m:n-1),g(1:m))      Compute numerator and denom-
    shn=-r(n+m1)+dot_product(r(n+m:n+1:-1),h(1:m))       inator for G and H,
    sgd=-r(n)+dot_product(r(n-m:n-1),h(m:1:-1))
    if (sd == 0.0 .or. sgd == 0.0) exit
    g(m1)=sgn/sgd                            whence G and H.
    h(m1)=shn/sd
    t(1:m)=g(1:m)
    g(1:m)=g(1:m)-g(m1)*h(m:1:-1)
    h(1:m)=h(1:m)-h(m1)*t(m:1:-1)
end do                                       Back for another recurrence.
if (m > n) call nrerror('toeplz: sanity check failed in routine')
call nrerror('toeplz: singular principal minor')
END FUNCTION toeplz
```

⋆ ⋆ ⋆

```
SUBROUTINE choldc(a,p)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: p
```

Given an $N \times N$ positive-definite symmetric matrix a, this routine constructs its Cholesky decomposition, $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$. On input, only the upper triangle of a need be given; it is not modified. The Cholesky factor $\mathbf{L}$ is returned in the lower triangle of a, except for its diagonal elements, which are returned in p, a vector of length N.

```
INTEGER(I4B) :: i,n
REAL(SP) :: summ
n=assert_eq(size(a,1),size(a,2),size(p),'choldc')
do i=1,n
```

```
    summ=a(i,i)-dot_product(a(i,1:i-1),a(i,1:i-1))
    if (summ <= 0.0) call nrerror('choldc failed')      a, with rounding errors, is
    p(i)=sqrt(summ)                                      not positive definite.
    a(i+1:n,i)=(a(i,i+1:n)-matmul(a(i+1:n,1:i-1),a(i,1:i-1)))/p(i)
end do
END SUBROUTINE choldc
```

```
SUBROUTINE cholsl(a,p,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: p,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
```

Solves the set of N linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where a is a positive-definite symmetric matrix. a ($N \times N$) and p (of length N) are input as the output of the routine choldc. Only the lower triangle of a is accessed. b is the input right-hand-side vector, of length N. The solution vector, also of length N, is returned in x. a and p are not modified and can be left in place for successive calls with different right-hand sides b. b is not modified unless you identify b and x in the calling sequence, which is allowed.

```
INTEGER(I4B) :: i,n
n=assert_eq((/size(a,1),size(a,2),size(p),size(b),size(x)/),'cholsl')
do i=1,n                          Solve L · y = b, storing y in x.
    x(i)=(b(i)-dot_product(a(i,1:i-1),x(1:i-1)))/p(i)
end do
do i=n,1,-1                       Solve L^T · x = y.
    x(i)=(x(i)-dot_product(a(i+1:n,i),x(i+1:n)))/p(i)
end do
END SUBROUTINE cholsl
```

$\star \quad \star \quad \star$

```
SUBROUTINE qrdcmp(a,c,d,sing)
USE nrtype; USE nrutil, ONLY : assert_eq,outerprod,vabs
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: c,d
LOGICAL(LGT), INTENT(OUT) :: sing
```

Constructs the QR decomposition of the $n \times n$ matrix a. The upper triangular matrix $\mathbf{R}$ is returned in the upper triangle of a, except for the diagonal elements of $\mathbf{R}$, which are returned in the n-dimensional vector d. The orthogonal matrix $\mathbf{Q}$ is represented as a product of $n-1$ Householder matrices $\mathbf{Q}_1 \ldots \mathbf{Q}_{n-1}$, where $\mathbf{Q}_j = \mathbf{1} - \mathbf{u}_j \otimes \mathbf{u}_j / c_j$. The ith component of $\mathbf{u}_j$ is zero for $i = 1, \ldots, j-1$ while the nonzero components are returned in a(i,j) for $i = j, \ldots, n$. sing returns as true if singularity is encountered during the decomposition, but the decomposition is still completed in this case.

```
INTEGER(I4B) :: k,n
REAL(SP) :: scale,sigma
n=assert_eq(size(a,1),size(a,2),size(c),size(d),'qrdcmp')
sing=.false.
do k=1,n-1
    scale=maxval(abs(a(k:n,k)))
    if (scale == 0.0) then          Singular case.
        sing=.true.
        c(k)=0.0
        d(k)=0.0
    else                            Form Q_k and Q_k · A.
        a(k:n,k)=a(k:n,k)/scale
        sigma=sign(vabs(a(k:n,k)),a(k,k))
        a(k,k)=a(k,k)+sigma
        c(k)=sigma*a(k,k)
```

```
        d(k)=-scale*sigma
        a(k:n,k+1:n)=a(k:n,k+1:n)-outerprod(a(k:n,k),&
            matmul(a(k:n,k),a(k:n,k+1:n)))/c(k)
    end if
end do
d(n)=a(n,n)
if (d(n) == 0.0) sing=.true.
END SUBROUTINE qrdcmp
```

`a(k:n,k+1:n)=a(k:n,k+1:n)-outerprod...matmul...` See discussion of equation (22.1.6).

```
SUBROUTINE qrsolv(a,c,d,b)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : rsolv
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: c,d
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Solves the set of n linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. The $n \times n$ matrix a and the n-dimensional vectors c and d are input as the output of the routine qrdcmp and are not modified. b is input as the right-hand-side vector of length n, and is overwritten with the solution vector on output.

```
INTEGER(I4B) :: j,n
REAL(SP) :: tau
n=assert_eq((/size(a,1),size(a,2),size(b),size(c),size(d)/),'qrsolv')
do j=1,n-1                      Form Q^T · b.
    tau=dot_product(a(j:n,j),b(j:n))/c(j)
    b(j:n)=b(j:n)-tau*a(j:n,j)
end do
call rsolv(a,d,b)               Solve R · x = Q^T · b.
END SUBROUTINE qrsolv
```

```
SUBROUTINE rsolv(a,d,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
```

Solves the set of n linear equations $\mathbf{R} \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{R}$ is an upper triangular matrix stored in a and d. The $n \times n$ matrix a and the vector d of length n are input as the output of the routine qrdcmp and are not modified. b is input as the right-hand-side vector of length n, and is overwritten with the solution vector on output.

```
INTEGER(I4B) :: i,n
n=assert_eq(size(a,1),size(a,2),size(b),size(d),'rsolv')
b(n)=b(n)/d(n)
do i=n-1,1,-1
    b(i)=(b(i)-dot_product(a(i,i+1:n),b(i+1:n)))/d(i)
end do
END SUBROUTINE rsolv
```

⋆ ⋆ ⋆

```
SUBROUTINE qrupdt(r,qt,u,v)
USE nrtype; USE nrutil, ONLY : assert_eq,ifirstloc
USE nr, ONLY : rotate,pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: r,qt
REAL(SP), DIMENSION(:), INTENT(INOUT) :: u
REAL(SP), DIMENSION(:), INTENT(IN) :: v
```

Given the QR decomposition of some $n \times n$ matrix, calculates the QR decomposition of the matrix $\mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v})$. Here `r` and `qt` are $n \times n$ matrices, `u` and `v` are n-dimensional vectors. Note that $\mathbf{Q}^T$ is input and returned in `qt`.

```
INTEGER(I4B) :: i,k,n
n=assert_eq((/size(r,1),size(r,2),size(qt,1),size(qt,2),size(u),&
    size(v)/),'qrupdt')
k=n+1-ifirstloc(u(n:1:-1) /= 0.0)          Find largest k such that u(k) ≠ 0.
if (k < 1) k=1
do i=k-1,1,-1                              Transform R + u ⊗ v to upper Hessenberg.
    call rotate(r,qt,i,u(i),-u(i+1))
    u(i)=pythag(u(i),u(i+1))
end do
r(1,:)=r(1,:)+u(1)*v
do i=1,k-1                                 Transform upper Hessenberg matrix to upper
    call rotate(r,qt,i,r(i,i),-r(i+1,i))       triangular.
end do
END SUBROUTINE qrupdt
```

f90 `k=n+1-ifirstloc(u(n:1:-1) /= 0.0)` The function `ifirstloc` in `nrutil` returns the first occurrence of `.true.` in a logical vector. See the discussion of the analogous routine `imaxloc` on p. 1017.

```
SUBROUTINE rotate(r,qt,i,a,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), TARGET, INTENT(INOUT) :: r,qt
INTEGER(I4B), INTENT(IN) :: i
REAL(SP), INTENT(IN) :: a,b
```

Given $n \times n$ matrices `r` and `qt`, carry out a Jacobi rotation on rows `i` and `i`$+1$ of each matrix. `a` and `b` are the parameters of the rotation: $\cos\theta = a/\sqrt{a^2+b^2}$, $\sin\theta = b/\sqrt{a^2+b^2}$.

```
REAL(SP), DIMENSION(size(r,1)) :: temp
INTEGER(I4B) :: n
REAL(SP) :: c,fact,s
n=assert_eq(size(r,1),size(r,2),size(qt,1),size(qt,2),'rotate')
if (a == 0.0) then                 Avoid unnecessary overflow or underflow.
    c=0.0
    s=sign(1.0_sp,b)
else if (abs(a) > abs(b)) then
    fact=b/a
    c=sign(1.0_sp/sqrt(1.0_sp+fact**2),a)
    s=fact*c
else
    fact=a/b
    s=sign(1.0_sp/sqrt(1.0_sp+fact**2),b)
    c=fact*s
end if
temp(i:n)=r(i,i:n)                 Premultiply r by Jacobi rotation.
r(i,i:n)=c*temp(i:n)-s*r(i+1,i:n)
r(i+1,i:n)=s*temp(i:n)+c*r(i+1,i:n)
temp=qt(i,:)                       Premultiply qt by Jacobi rotation.
qt(i,:)=c*temp-s*qt(i+1,:)
qt(i+1,:)=s*temp+c*qt(i+1,:)
END SUBROUTINE rotate
```

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press). [1]

Gu, M., Demmel, J., and Dhillon, I. 1994, LAPACK Working Note #88 (Computer Science Department, University of Tennessee at Knoxville, Preprint UT-CS-94-257; available from Netlib, or as http://www.cs.utk.edu/∼library/TechReports/1994/ut-cs-94-257.ps.Z). [2] See also discussion after `tqli` in Chapter B11.

References

The references collected here are those of general usefulness, cited in this volume. For references to the material in Volume 1, see the References section of that volume.

A first group of references relates to the Fortran 90 language itself:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

Kerrigan, J.F. 1993, *Migrating to Fortran 90* (Sebastopol, CA: O'Reilly).

Brainerd, W.S., Goldberg, C.H., and Adams, J.C. 1996, *Programmer's Guide to Fortran 90*, 3rd ed. (New York: Springer-Verlag).

A second group of references relates to, or includes material on, parallel programming and algorithms:

Akl, S.G. 1989, *The Design and Analysis of Parallel Algorithms* (Englewood Cliffs, NJ: Prentice Hall).

Bertsekas, D.P., and Tsitsiklis, J.N. 1989, *Parallel and Distributed Computation: Numerical Methods* (Englewood Cliffs, NJ: Prentice Hall).

Carey, G.F. 1989, *Parallel Supercomputing: Methods, Algorithms, and Applications* (New York: Wiley).

Fountain, T.J. 1994, *Parallel Computing: Principles and Practice* (New York: Cambridge University Press).

Fox, G.C., et al. 1988, *Solving Problems on Concurrent Processors*, Volume I (Englewood Cliffs, NJ: Prentice Hall).

Golub, G., and Ortega, J.M. 1993, *Scientific Computing: An Introduction with Parallel Computing* (San Diego, CA: Academic Press).

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press).

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2* (Bristol and Philadelphia: Adam Hilger).

Kumar, V., et al. 1994, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms* (Redwood City, CA: Benjamin/Cummings).

Lewis, T.G., and El-Rewini, H. 1992, *Introduction to Parallel Computing* (Englewood Cliffs, NJ: Prentice Hall).

Modi, J.J. 1988, *Parallel Algorithms and Matrix Computation* (New York: Oxford University Press).

Smith, J.R. 1993, *The Design and Analysis of Parallel Algorithms* (New York: Oxford University Press).

Van de Velde, E. 1994, *Concurrent Scientific Computing* (New York: Springer-Verlag).

Van Loan, C.F. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.).

Chapter B3. Interpolation and Extrapolation

```
SUBROUTINE polint(xa,ya,x,y,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
```

Given arrays xa and ya of length N, and given a value x, this routine returns a value y, and an error estimate dy. If $P(x)$ is the polynomial of degree $N-1$ such that $P(\mathtt{xa}_i) = \mathtt{ya}_i, i = 1,\ldots,N$, then the returned value $\mathtt{y} = P(\mathtt{x})$.

```
INTEGER(I4B) :: m,n,ns
REAL(SP), DIMENSION(size(xa)) :: c,d,den,ho
n=assert_eq(size(xa),size(ya),'polint')
c=ya                                        Initialize the tableau of c's and d's.
d=ya
ho=xa-x
ns=iminloc(abs(x-xa))                       Find index ns of closest table entry.
y=ya(ns)                                    This is the initial approximation to y.
ns=ns-1
do m=1,n-1                                  For each column of the tableau,
    den(1:n-m)=ho(1:n-m)-ho(1+m:n)          we loop over the current c's and d's and up-
    if (any(den(1:n-m) == 0.0)) &               date them.
        call nrerror('polint: calculation failure')
```

This error can occur only if two input xa's are (to within roundoff) identical.

```
    den(1:n-m)=(c(2:n-m+1)-d(1:n-m))/den(1:n-m)
    d(1:n-m)=ho(1+m:n)*den(1:n-m)           Here the c's and d's are updated.
    c(1:n-m)=ho(1:n-m)*den(1:n-m)
    if (2*ns < n-m) then                    After each column in the tableau is completed, we decide
        dy=c(ns+1)                              which correction, c or d, we want to add to our accu-
    else                                        mulating value of y, i.e., which path to take through
        dy=d(ns)                                the tableau—forking up or down. We do this in such a
        ns=ns-1                                 way as to take the most "straight line" route through the
    end if                                      tableau to its apex, updating ns accordingly to keep track
    y=y+dy                                      of where we are. This route keeps the partial approxima-
end do                                          tions centered (insofar as possible) on the target x. The
END SUBROUTINE polint                           last dy added is thus the error indication.
```

```
SUBROUTINE ratint(xa,ya,x,y,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: y,dy
```

Given arrays xa and ya of length N, and given a value of x, this routine returns a value of y and an accuracy estimate dy. The value returned is that of the diagonal rational function, evaluated at x, that passes through the N points $(\mathtt{xa}_i, \mathtt{ya}_i)$, $i = 1 \ldots N$.

```
INTEGER(I4B) :: m,n,ns
REAL(SP), DIMENSION(size(xa)) :: c,d,dd,h,t
```

```
REAL(SP), PARAMETER :: TINY=1.0e-25_sp          A small number.
n=assert_eq(size(xa),size(ya),'ratint')
h=xa-x
ns=iminloc(abs(h))
y=ya(ns)
if (x == xa(ns)) then
    dy=0.0
    RETURN
end if
c=ya
d=ya+TINY                                       The TINY part is needed to prevent
ns=ns-1                                             a rare zero-over-zero condition.
do m=1,n-1
    t(1:n-m)=(xa(1:n-m)-x)*d(1:n-m)/h(1+m:n)    h will never be zero, since this was
    dd(1:n-m)=t(1:n-m)-c(2:n-m+1)                   tested in the initializing loop.
    if (any(dd(1:n-m) == 0.0)) &
        call nrerror('failure in ratint')       This error condition indicates that
    dd(1:n-m)=(c(2:n-m+1)-d(1:n-m))/dd(1:n-m)       the interpolating function has a
    d(1:n-m)=c(2:n-m+1)*dd(1:n-m)                   pole at the requested value of
    c(1:n-m)=t(1:n-m)*dd(1:n-m)                     x.
    if (2*ns < n-m) then
        dy=c(ns+1)
    else
        dy=d(ns)
        ns=ns-1
    end if
    y=y+dy
end do
END SUBROUTINE ratint
```

⋆ ⋆ ⋆

```
SUBROUTINE spline(x,y,yp1,ypn,y2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : tridag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(IN) :: yp1,ypn
REAL(SP), DIMENSION(:), INTENT(OUT) :: y2
```

Given arrays x and y of length N containing a tabulated function, i.e., $\mathtt{y}_i = f(\mathtt{x}_i)$, with $\mathtt{x}_1 < \mathtt{x}_2 < \ldots < \mathtt{x}_N$, and given values yp1 and ypn for the first derivative of the interpolating function at points 1 and N, respectively, this routine returns an array y2 of length N that contains the second derivatives of the interpolating function at the tabulated points $\mathtt{x}_i$. If yp1 and/or ypn are equal to 1×10^{30} or larger, the routine is signaled to set the corresponding boundary condition for a natural spline, with zero second derivative on that boundary.

```
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(x)) :: a,b,c,r
n=assert_eq(size(x),size(y),size(y2),'spline')
c(1:n-1)=x(2:n)-x(1:n-1)                    Set up the tridiagonal equations.
r(1:n-1)=6.0_sp*((y(2:n)-y(1:n-1))/c(1:n-1))
r(2:n-1)=r(2:n-1)-r(1:n-2)
a(2:n-1)=c(1:n-2)
b(2:n-1)=2.0_sp*(c(2:n-1)+a(2:n-1))
b(1)=1.0
b(n)=1.0
if (yp1 > 0.99e30_sp) then                  The lower boundary condition is set either to be "nat-
    r(1)=0.0                                    ural"
    c(1)=0.0
else                                        or else to have a specified first derivative.
    r(1)=(3.0_sp/(x(2)-x(1)))*((y(2)-y(1))/(x(2)-x(1))-yp1)
```

```
    c(1)=0.5
end if                                  The upper boundary condition is set either to be
if (ypn > 0.99e30_sp) then                  "natural"
    r(n)=0.0
    a(n)=0.0
else                                    or else to have a specified first derivative.
    r(n)=(-3.0_sp/(x(n)-x(n-1)))*((y(n)-y(n-1))/(x(n)-x(n-1))-ypn)
    a(n)=0.5
end if
call tridag(a(2:n),b(1:n),c(1:n-1),r(1:n),y2(1:n))
END SUBROUTINE spline



FUNCTION splint(xa,ya,y2a,x)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY: locate
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya,y2a
REAL(SP), INTENT(IN) :: x
REAL(SP) :: splint
```

Given the arrays xa and ya, which tabulate a function (with the $\mathtt{xa}_i$'s in increasing or decreasing order), and given the array y2a, which is the output from spline above, and given a value of x, this routine returns a cubic-spline interpolated value. The arrays xa, ya and y2a are all of the same size.

```
INTEGER(I4B) :: khi,klo,n
REAL(SP) :: a,b,h
n=assert_eq(size(xa),size(ya),size(y2a),'splint')
klo=max(min(locate(xa,x),n-1),1)
```

We will find the right place in the table by means of locate's bisection algorithm. This is optimal if sequential calls to this routine are at random values of x. If sequential calls are in order, and closely spaced, one would do better to store previous values of klo and khi and test if they remain appropriate on the next call.

```
khi=klo+1                       klo and khi now bracket the input value of x.
h=xa(khi)-xa(klo)
if (h == 0.0) call nrerror('bad xa input in splint')    The xa's must be distinct.
a=(xa(khi)-x)/h                 Cubic spline polynomial is now evaluated.
b=(x-xa(klo))/h
splint=a*ya(klo)+b*ya(khi)+((a**3-a)*y2a(klo)+(b**3-b)*y2a(khi))*(h**2)/6.0_sp
END FUNCTION splint
```

f90 `klo=max(min(locate(xa,x),n-1),1)` In the Fortran 77 version of `splint`, there is in-line code to find the location in the table by bisection. Here we prefer an explicit call to `locate`, which performs the bisection. On some massively multiprocessor (MMP) machines, one might substitute a different, more parallel algorithm (see next note).

⋆ ⋆ ⋆

```
FUNCTION locate(xx,x)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
REAL(SP), INTENT(IN) :: x
INTEGER(I4B) :: locate
```

Given an array $\mathtt{xx}(1{:}N)$, and given a value x, returns a value j such that x is between $\mathtt{xx}(j)$ and $\mathtt{xx}(j+1)$. xx must be monotonic, either increasing or decreasing. $j = 0$ or $j = N$ is returned to indicate that x is out of range.

```
INTEGER(I4B) :: n,jl,jm,ju
LOGICAL :: ascnd
```

```
n=size(xx)
ascnd = (xx(n) >= xx(1))          True if ascending order of table, false otherwise.
jl=0                              Initialize lower
ju=n+1                            and upper limits.
do
    if (ju-jl <= 1) exit          Repeat until this condition is satisfied.
    jm=(ju+jl)/2                  Compute a midpoint,
    if (ascnd .eqv. (x >= xx(jm))) then
        jl=jm                     and replace either the lower limit
    else
        ju=jm                     or the upper limit, as appropriate.
    end if
end do
if (x == xx(1)) then              Then set the output, being careful with the endpoints.
    locate=1
else if (x == xx(n)) then
    locate=n-1
else
    locate=jl
end if
END FUNCTION locate
```

The use of bisection is perhaps a sin on a genuinely parallel machine, but (since the process takes only logarithmically many sequential steps) it is at most a *small* sin. One can imagine a "fully parallel" implementation like,

```
k=iminloc(abs(x-xx))
if ((x < xx(k)) .eqv. (xx(1) < xx(n))) then
        locate=k-1
else
        locate=k
end if
```

Problem is, unless the number of *physical* (not logical) processors participating in the `iminloc` is larger than N, the length of the array, this "parallel" code turns a $\log N$ algorithm into one scaling as N, quite an unacceptable inefficiency. So we prefer to be small sinners and bisect.

```
SUBROUTINE hunt(xx,x,jlo)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: jlo
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
```

Given an array `xx(1:`N`)`, and given a value `x`, returns a value `jlo` such that `x` is between `xx(jlo)` and `xx(jlo+1)`. `xx` must be monotonic, either increasing or decreasing. `jlo` $= 0$ or `jlo` $= N$ is returned to indicate that `x` is out of range. `jlo` on input is taken as the initial guess for `jlo` on output.

```
INTEGER(I4B) :: n,inc,jhi,jm
LOGICAL :: ascnd
n=size(xx)
ascnd = (xx(n) >= xx(1))              True if ascending order of table, false otherwise.
if (jlo <= 0 .or. jlo > n) then       Input guess not useful. Go immediately to bisec-
    jlo=0                                 tion.
    jhi=n+1
else
    inc=1                             Set the hunting increment.
    if (x >= xx(jlo) .eqv. ascnd) then        Hunt up:
        do
            jhi=jlo+inc
            if (jhi > n) then         Done hunting, since off end of table.
```

```
            jhi=n+1
            exit
        else
            if (x < xx(jhi) .eqv. ascnd) exit
            jlo=jhi                 Not done hunting,
            inc=inc+inc             so double the increment
        end if
    end do                          and try again.
else                                Hunt down:
    jhi=jlo
    do
        jlo=jhi-inc
        if (jlo < 1) then           Done hunting, since off end of table.
            jlo=0
            exit
        else
            if (x >= xx(jlo) .eqv. ascnd) exit
            jhi=jlo                 Not done hunting,
            inc=inc+inc             so double the increment
        end if
    end do                          and try again.
end if
end if                              Done hunting, value bracketed.
do                                  Hunt is done, so begin the final bisection phase:
    if (jhi-jlo <= 1) then
        if (x == xx(n)) jlo=n-1
        if (x == xx(1)) jlo=1
        exit
    else
        jm=(jhi+jlo)/2
        if (x >= xx(jm) .eqv. ascnd) then
            jlo=jm
        else
            jhi=jm
        end if
    end if
end do
END SUBROUTINE hunt
```

⋆ ⋆ ⋆

```
FUNCTION polcoe(x,y)
USE nrtype; USE nrutil, ONLY : assert_eq,outerdiff
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(size(x)) :: polcoe
```

Given same-size arrays x and y containing a tabulated function $\mathtt{y}_i = f(\mathtt{x}_i)$, this routine returns a same-size array of coefficients c_j, such that $\mathtt{y}_i = \sum_j c_j \mathtt{x}_i^{j-1}$.

```
INTEGER(I4B) :: i,k,n
REAL(SP), DIMENSION(size(x)) :: s
REAL(SP), DIMENSION(size(x),size(x)) :: a
n=assert_eq(size(x),size(y),'polcoe')
s=0.0                           Coefficients s_i of the master polynomial P(x) are found by
s(n)=-x(1)                          recurrence.
do i=2,n
    s(n+1-i:n-1)=s(n+1-i:n-1)-x(i)*s(n+2-i:n)
    s(n)=s(n)-x(i)
end do
a=outerdiff(x,x)                Make vector w_j = ∏_{j≠n}(x_j − x_n), using polcoe for tempo-
polcoe=product(a,dim=2,mask=a /= 0.0)   rary storage.
```

```
  Now do synthetic division by x − x_j. The division for all x_j can be done in parallel (on a
  parallel machine), since the : in the loop below is over j.
a(:,1)=-s(1)/x(:)
do k=2,n
    a(:,k)=-(s(k)-a(:,k-1))/x(:)
end do
s=y/polcoe
polcoe=matmul(s,a)          Solve linear system.
END FUNCTION polcoe
```

For a description of the coding here, see §22.3, especially equation (22.3.9). You might also want to compare the coding here with the Fortran 77 version, and also look at the description of the method on p. 84 in Volume 1. The Fortran 90 implementation here is in fact much closer to that description than is the Fortran 77 method, which goes through some acrobatics to roll the synthetic division and matrix multiplication into a single set of two nested loops. The price we pay, here, is storage for the matrix a. Since the degree of any useful polynomial is not a very large number, this is essentially no penalty.

Also worth noting is the way that parallelism is brought to the required synthetic division. For a *single* such synthetic division (e.g., as accomplished by the nrutil routine poly_term), parallelism can be obtained only by recursion. Here things are much simpler, because we need a whole bunch of simultaneous and independent synthetic divisions; so we can just do them in the obvious, data-parallel, way.

```
FUNCTION polcof(xa,ya)
USE nrtype; USE nrutil, ONLY : assert_eq,iminloc
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
REAL(SP), DIMENSION(size(xa)) :: polcof
    Given same-size arrays xa and ya containing a tabulated function ya_i = f(xa_i), this routine
    returns a same-size array of coefficients c_j such that ya_i = Σ_j c_j xa_i^{j−1}.
INTEGER(I4B) :: j,k,m,n
REAL(SP) :: dy
REAL(SP), DIMENSION(size(xa)) :: x,y
n=assert_eq(size(xa),size(ya),'polcof')
x=xa
y=ya
do j=1,n
    m=n+1-j
    call polint(x(1:m),y(1:m),0.0_sp,polcof(j),dy)
      Use the polynomial interpolation routine of §3.1 to extrapolate to x = 0.
    k=iminloc(abs(x(1:m)))              Find the remaining x_k of smallest absolute value,
    where (x(1:m) /= 0.0) y(1:m)=(y(1:m)-polcof(j))/x(1:m)      reduce all the terms,
    y(k:m-1)=y(k+1:m)                   and eliminate x_k.
    x(k:m-1)=x(k+1:m)
end do
END FUNCTION polcof
```

⋆ ⋆ ⋆

```
SUBROUTINE polin2(x1a,x2a,ya,x1,x2,y,dy)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: y,dy
```

Given arrays x1a of length M and x2a of length N of independent variables, and an $M \times N$ array of function values ya, tabulated at the grid points defined by x1a and x2a, and given values x1 and x2 of the independent variables, this routine returns an interpolated function value y, and an accuracy indication dy (based only on the interpolation in the x1 direction, however).

```
INTEGER(I4B) :: j,m,ndum
REAL(SP), DIMENSION(size(x1a)) :: ymtmp
REAL(SP), DIMENSION(size(x2a)) :: yntmp
m=assert_eq(size(x1a),size(ya,1),'polin2: m')
ndum=assert_eq(size(x2a),size(ya,2),'polin2: ndum')
do j=1,m                                        Loop over rows.
    yntmp=ya(j,:)                               Copy row into temporary storage.
    call polint(x2a,yntmp,x2,ymtmp(j),dy)       Interpolate answer into temporary stor-
end do                                              age.
call polint(x1a,ymtmp,x1,y,dy)                  Do the final interpolation.
END SUBROUTINE polin2
```

$\star \quad \star \quad \star$

```
SUBROUTINE bcucof(y,y1,y2,y12,d1,d2,c)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: d1,d2
REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
REAL(SP), DIMENSION(4,4), INTENT(OUT) :: c
```

Given arrays y, y1, y2, and y12, each of length 4, containing the function, gradients, and cross derivative at the four grid points of a rectangular grid cell (numbered counterclockwise from the lower left), and given d1 and d2, the length of the grid cell in the 1- and 2-directions, this routine returns the 4×4 table c that is used by routine bcuint for bicubic interpolation.

```
REAL(SP), DIMENSION(16) :: x
REAL(SP), DIMENSION(16,16) :: wt
DATA wt /1,0,-3,2,4*0,-3,0,9,-6,2,0,-6,4,&
    8*0,3,0,-9,6,-2,0,6,-4,10*0,9,-6,2*0,-6,4,2*0,3,-2,6*0,-9,6,&
    2*0,6,-4,4*0,1,0,-3,2,-2,0,6,-4,1,0,-3,2,8*0,-1,0,3,-2,1,0,-3,&
    2,10*0,-3,2,2*0,3,-2,6*0,3,-2,2*0,-6,4,2*0,3,-2,0,1,-2,1,5*0,&
    -3,6,-3,0,2,-4,2,9*0,3,-6,3,0,-2,4,-2,10*0,-3,3,2*0,2,-2,2*0,&
    -1,1,6*0,3,-3,2*0,-2,2,5*0,1,-2,1,0,-2,4,-2,0,1,-2,1,9*0,-1,2,&
    -1,0,1,-2,1,10*0,1,-1,2*0,-1,1,6*0,-1,1,2*0,2,-2,2*0,-1,1/
x(1:4)=y                                        Pack a temporary vector x.
x(5:8)=y1*d1
x(9:12)=y2*d2
x(13:16)=y12*d1*d2
x=matmul(wt,x)                                  Matrix multiply by the stored table.
c=reshape(x,(/4,4/),order=(/2,1/))              Unpack the result into the output table.
END SUBROUTINE bcucof
```

f90 `x=matmul(wt,x) ... c=reshape(x,(/4,4/),order=(/2,1/))` It is a powerful technique to combine the `matmul` intrinsic with `reshape`'s of the input or output. The idea is to use `matmul` whenever the calculation can be cast into the form of a linear mapping between input and output objects. Here the `order=(/2,1/)` parameter specifies that we want the packing to be by rows, not by Fortran's default of columns. (In this two-dimensional case, it's the equivalent of applying `transpose`.)

```
SUBROUTINE bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,ansy1,ansy2)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : bcucof
IMPLICIT NONE
REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
REAL(SP), INTENT(IN) :: x1l,x1u,x2l,x2u,x1,x2
REAL(SP), INTENT(OUT) :: ansy,ansy1,ansy2
```

Bicubic interpolation within a grid square. Input quantities are `y,y1,y2,y12` (as described in `bcucof`); `x1l` and `x1u`, the lower and upper coordinates of the grid square in the 1-direction; `x2l` and `x2u` likewise for the 2-direction; and `x1,x2`, the coordinates of the desired point for the interpolation. The interpolated function value is returned as `ansy`, and the interpolated gradient values as `ansy1` and `ansy2`. This routine calls `bcucof`.

```
INTEGER(I4B) :: i
REAL(SP) :: t,u
REAL(SP), DIMENSION(4,4) :: c
call bcucof(y,y1,y2,y12,x1u-x1l,x2u-x2l,c)        Get the c's.
if (x1u == x1l .or. x2u == x2l) call &
    nrerror('bcuint: problem with input values - boundary pair equal?')
t=(x1-x1l)/(x1u-x1l)                              Equation (3.6.4).
u=(x2-x2l)/(x2u-x2l)
ansy=0.0
ansy2=0.0
ansy1=0.0
do i=4,1,-1                                       Equation (3.6.6).
    ansy=t*ansy+((c(i,4)*u+c(i,3))*u+c(i,2))*u+c(i,1)
    ansy2=t*ansy2+(3.0_sp*c(i,4)*u+2.0_sp*c(i,3))*u+c(i,2)
    ansy1=u*ansy1+(3.0_sp*c(4,i)*t+2.0_sp*c(3,i))*t+c(2,i)
end do
ansy1=ansy1/(x1u-x1l)
ansy2=ansy2/(x2u-x2l)
END SUBROUTINE bcuint
```

⋆ ⋆ ⋆

```
SUBROUTINE splie2(x1a,x2a,ya,y2a)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : spline
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: y2a
```

Given an $M \times N$ tabulated function `ya`, and N tabulated independent variables `x2a`, this routine constructs one-dimensional natural cubic splines of the rows of `ya` and returns the second derivatives in the $M \times N$ array `y2a`. (The array `x1a` is included in the argument list merely for consistency with routine `splin2`.)

```
INTEGER(I4B) :: j,m,ndum
m=assert_eq(size(x1a),size(ya,1),size(y2a,1),'splie2: m')
ndum=assert_eq(size(x2a),size(ya,2),size(y2a,2),'splie2: ndum')
do j=1,m
    call spline(x2a,ya(j,:),1.0e30_sp,1.0e30_sp,y2a(j,:))
```

Values 1×10^{30} signal a natural spline.

```
end do
END SUBROUTINE splie2
```

```
FUNCTION splin2(x1a,x2a,ya,y2a,x1,x2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : spline,splint
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya,y2a
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP) :: splin2
```

Given x1a, x2a, ya as described in splie2 and y2a as produced by that routine; and given a desired interpolating point x1,x2; this routine returns an interpolated function value by bicubic spline interpolation.

```
INTEGER(I4B) :: j,m,ndum
REAL(SP), DIMENSION(size(x1a)) :: yytmp,y2tmp2
m=assert_eq(size(x1a),size(ya,1),size(y2a,1),'splin2: m')
ndum=assert_eq(size(x2a),size(ya,2),size(y2a,2),'splin2: ndum')
do j=1,m
    yytmp(j)=splint(x2a,ya(j,:),y2a(j,:),x2)
```

Perform m evaluations of the row splines constructed by splie2, using the one-dimensional spline evaluator splint.

```
end do
call spline(x1a,yytmp,1.0e30_sp,1.0e30_sp,y2tmp2)
```

Construct the one-dimensional column spline and evaluate it.

```
splin2=splint(x1a,yytmp,y2tmp2,x1)
END FUNCTION splin2
```

Chapter B4. Integration of Functions

```
SUBROUTINE trapzd(func,a,b,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
```

This routine computes the nth stage of refinement of an extended trapezoidal rule. `func` is input as the name of the function to be integrated between limits `a` and `b`, also input. When called with `n`=1, the routine returns as `s` the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls with `n`=2,3,... (in that sequential order) will improve the accuracy of `s` by adding 2^{n-2} additional interior points. `s` should not be modified between sequential calls.

```
REAL(SP) :: del,fsum
INTEGER(I4B) :: it
if (n == 1) then
    s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))
else
    it=2**(n-2)
    del=(b-a)/it                    This is the spacing of the points to be added.
    fsum=sum(func(arth(a+0.5_sp*del,del,it)))
    s=0.5_sp*(s+del*fsum)           This replaces s by its refined value.
end if
END SUBROUTINE trapzd
```

While most of the quadrature routines in this chapter are coded as functions, `trapzd` is a subroutine because the argument `s` that returns the function value must also be supplied as an input parameter. We could change the subroutine into a function by declaring `s` to be a local variable with the `SAVE` attribute. However, this would prevent us from being able to use the routine recursively to do multidimensional quadrature (see `quad3d` on p. 1065). When `s` is left as an argument, a fresh copy is created on each recursive call. As a `SAVE`'d variable, by contrast, its value would get overwritten on each call, and the code would not be properly "re-entrant."

`s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))` Note how we use the `(/.../)` construct to supply a set of scalar arguments to a vector function.

⋆ ⋆ ⋆

```
FUNCTION qtrap(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qtrap
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20
REAL(SP), PARAMETER :: EPS=1.0e-6_sp, UNLIKELY=-1.0e30_sp
```

Returns the integral of the function func from a to b. The parameter EPS should be set to the desired fractional accuracy and JMAX so that 2 to the power JMAX−1 is the maximum allowed number of steps. Integration is performed by the trapezoidal rule.

```
REAL(SP) :: olds
INTEGER(I4B) :: j
olds=UNLIKELY                          Any number that is unlikely to be the average of the
do j=1,JMAX                                function at its endpoints will do here.
    call trapzd(func,a,b,qtrap,j)
    if (j > 5) then                    Avoid spurious early convergence.
        if (abs(qtrap-olds) < EPS*abs(olds) .or. &
            (qtrap == 0.0 .and. olds == 0.0)) RETURN
    end if
    olds=qtrap
end do
call nrerror('qtrap: too many steps')
END FUNCTION qtrap
```

★ ★ ★

```
FUNCTION qsimp(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qsimp
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20
REAL(SP), PARAMETER :: EPS=1.0e-6_sp, UNLIKELY=-1.0e30_sp
```

Returns the integral of the function func from a to b. The parameter EPS should be set to the desired fractional accuracy and JMAX so that 2 to the power JMAX−1 is the maximum allowed number of steps. Integration is performed by Simpson's rule.

```
INTEGER(I4B) :: j
REAL(SP) :: os,ost,st
ost=UNLIKELY
os= UNLIKELY
do j=1,JMAX
    call trapzd(func,a,b,st,j)
    qsimp=(4.0_sp*st-ost)/3.0_sp       Compare equation (4.2.4).
    if (j > 5) then                    Avoid spurious early convergence.
        if (abs(qsimp-os) < EPS*abs(os) .or. &
            (qsimp == 0.0 .and. os == 0.0)) RETURN
    end if
    os=qsimp
```

```
    ost=st
end do
call nrerror('qsimp: too many steps')
END FUNCTION qsimp
```

★ ★ ★

```
FUNCTION qromb(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint,trapzd
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromb
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20,JMAXP=JMAX+1,K=5,KM=K-1
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Returns the integral of the function `func` from `a` to `b`. Integration is performed by Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.

Parameters: EPS is the fractional accuracy desired, as determined by the extrapolation error estimate; JMAX limits the total number of steps; K is the number of points used in the extrapolation.

```
REAL(SP), DIMENSION(JMAXP) :: h,s
REAL(SP) :: dqromb
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
    call trapzd(func,a,b,s(j),j)
    if (j >= K) then
        call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromb,dqromb)
        if (abs(dqromb) <= EPS*abs(qromb)) RETURN
    end if
    s(j+1)=s(j)
    h(j+1)=0.25_sp*h(j)
end do
call nrerror('qromb: too many steps')
END FUNCTION qromb
```

These store the successive trapezoidal approximations and their relative stepsizes.

This is a key step: The factor is 0.25 even though the stepsize is decreased by only 0.5. This makes the extrapolation a polynomial in h^2 as allowed by equation (4.2.1), not just a polynomial in h.

★ ★ ★

```
SUBROUTINE midpnt(func,a,b,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
```

This routine computes the `n`th stage of refinement of an extended midpoint rule. `func` is input as the name of the function to be integrated between limits `a` and `b`, also input. When

called with n=1, the routine returns as s the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls with n=2,3,... (in that sequential order) will improve the accuracy of s by adding $(2/3) \times 3^{n-1}$ additional interior points. s should not be modified between sequential calls.

```
REAL(SP) :: del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
if (n == 1) then
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)              The added points alternate in spacing between
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)      del and 2*del.
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))        The new sum is combined with the old integral
end if                                     to give a refined integral.
END SUBROUTINE midpnt
```

midpnt is a subroutine and not a function for the same reasons as trapzd. This is also true for the other mid... routines below.

`s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))` Here we use `(/.../)` to pass a single scalar argument to a vector function.

⋆ ⋆ ⋆

```
FUNCTION qromo(func,a,b,choose)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromo
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func

    SUBROUTINE choose(funk,aa,bb,s,n)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: aa,bb
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
        FUNCTION funk(x)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        REAL(SP), DIMENSION(size(x)) :: funk
        END FUNCTION funk
    END INTERFACE
    END SUBROUTINE choose
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=14,JMAXP=JMAX+1,K=5,KM=K-1
REAL(SP), PARAMETER :: EPS=1.0e-6
```

Romberg integration on an open interval. Returns the integral of the function func from a to b, using any specified integrating subroutine choose and Romberg's method. Normally choose will be an open formula, not evaluating the function at the endpoints. It is assumed that choose triples the number of steps on each call, and that its error series contains only

even powers of the number of steps. The routines midpnt, midinf, midsql, midsqu, and midexp are possible choices for choose. The parameters have the same meaning as in qromb.

```
REAL(SP), DIMENSION(JMAXP) :: h,s
REAL(SP) :: dqromo
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
    call choose(func,a,b,s(j),j)
    if (j >= K) then
        call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromo,dqromo)
        if (abs(dqromo) <= EPS*abs(qromo)) RETURN
    end if
    s(j+1)=s(j)
    h(j+1)=h(j)/9.0_sp          This is where the assumption of step tripling and an even
end do                              error series is used.
call nrerror('qromo: too many steps')
END FUNCTION qromo
```

⋆ ⋆ ⋆

```
SUBROUTINE midinf(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
```

This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement of the integral of funk from aa to bb, except that the function is evaluated at evenly spaced points in $1/x$ rather than in x. This allows the upper limit bb to be as large and positive as the computer allows, or the lower limit aa to be as large and negative, but not both. aa and bb must have the same sign.

```
REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
call assert(aa*bb > 0.0, 'midinf args')
b=1.0_sp/aa                     These two statements change the limits of integration ac-
a=1.0_sp/bb                         cordingly.
if (n == 1) then                From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)            This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=funk(1.0_sp/x)/x**2
    END FUNCTION func
END SUBROUTINE midinf
```

f90 FUNCTION func(x) The change of variable could have been effected by a statement function in midinf itself. However, the statement function is a Fortran 77 feature that is deprecated in Fortran 90 because it does not allow the benefits of having an explicit interface, i.e., a complete set of specification statements. Statement functions can always be coded as internal subprograms instead.

```
SUBROUTINE midsql(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
```

This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement of the integral of funk from aa to bb, except that it allows for an inverse square-root singularity in the integrand at the lower limit aa.

```
REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=sqrt(bb-aa)                    These two statements change the limits of integration ac-
a=0.0                                cordingly.
if (n == 1) then                 From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)             This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=2.0_sp*x*funk(aa+x**2)
    END FUNCTION func
END SUBROUTINE midsql
```

```
SUBROUTINE midsqu(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
```

This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement of the integral of funk from aa to bb, except that it allows for an inverse square-root singularity in the integrand at the upper limit bb.

```
REAL(SP) :: a,b,del
```

```
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=sqrt(bb-aa)                    These two statements change the limits of integration ac-
a=0.0                                cordingly.
if (n == 1) then                 From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)             This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=2.0_sp*x*funk(bb-x**2)
    END FUNCTION func
END SUBROUTINE midsqu


SUBROUTINE midexp(funk,aa,bb,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: aa,bb
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
END INTERFACE
    This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement
    of the integral of funk from aa to bb, except that bb is assumed to be infinite (value passed
    not actually used). It is assumed that the function funk decreases exponentially rapidly at
    infinity.
REAL(SP) :: a,b,del
INTEGER(I4B) :: it
REAL(SP), DIMENSION(2*3**(n-2)) :: x
b=exp(-aa)                       These two statements change the limits of integration ac-
a=0.0                                cordingly.
if (n == 1) then                 From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*sum(func( (/0.5_sp*(a+b)/) ))
else
    it=3**(n-2)
    del=(b-a)/(3.0_sp*it)
    x(1:2*it-1:2)=arth(a+0.5_sp*del,3.0_sp*del,it)
    x(2:2*it:2)=x(1:2*it-1:2)+2.0_sp*del
    s=s/3.0_sp+del*sum(func(x))
end if
CONTAINS
    FUNCTION func(x)             This internal function effects the change of variable.
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    func=funk(-log(x))/x
    END FUNCTION func
END SUBROUTINE midexp
```

⋆ ⋆ ⋆

```
SUBROUTINE gauleg(x1,x2,x,w)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-14_dp
```

Given the lower and upper limits of integration x1 and x2, this routine returns arrays x and w of length N containing the abscissas and weights of the Gauss-Legendre N-point quadrature formula. The parameter EPS is the relative precision. Note that internal computations are done in double precision.

```
INTEGER(I4B) :: its,j,m,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(DP) :: xl,xm
REAL(DP), DIMENSION((size(x)+1)/2) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
n=assert_eq(size(x),size(w),'gauleg')
m=(n+1)/2                                   The roots are symmetric in the interval,
xm=0.5_dp*(x2+x1)                              so we only have to find half of them.
xl=0.5_dp*(x2-x1)
z=cos(PI_D*(arth(1,1,m)-0.25_dp)/(n+0.5_dp))   Initial approximations to the roots.
unfinished=.true.
do its=1,MAXIT                              Newton's method carried out simultane-
    where (unfinished)                         ously on the roots.
        p1=1.0
        p2=0.0
    end where
    do j=1,n                                Loop up the recurrence relation to get
        where (unfinished)                     the Legendre polynomials evaluated
            p3=p2                              at z.
            p2=p1
            p1=((2.0_dp*j-1.0_dp)*z*p2-(j-1.0_dp)*p3)/j
        end where
    end do
      p1 now contains the desired Legendre polynomials. We next compute pp, the derivatives,
      by a standard relation involving also p2, the polynomials of one lower order.
    where (unfinished)
        pp=n*(z*p1-p2)/(z*z-1.0_dp)
        z1=z
        z=z1-p1/pp                          Newton's method.
        unfinished=(abs(z-z1) > EPS)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gauleg')
x(1:m)=xm-xl*z                              Scale the root to the desired interval,
x(n:n-m+1:-1)=xm+xl*z                       and put in its symmetric counterpart.
w(1:m)=2.0_dp*xl/((1.0_dp-z**2)*pp**2)      Compute the weight
w(n:n-m+1:-1)=w(1:m)                        and its symmetric counterpart.
END SUBROUTINE gauleg
```

f90 Often we have an iterative procedure that has to be applied until all components of a vector have satisfied a convergence criterion. Some components of the vector might converge sooner than others, and it is inefficient on a small-scale parallel (SSP) machine to continue iterating on those components. The general structure we use for such an iteration is exemplified by the following lines from gauleg:

```
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
   ...
unfinished=.true.
do its=1,MAXIT
```

```
        where (unfinished)
              ...
           unfinished=(abs(z-z1) > EPS)
        end where
        if (.not. any(unfinished)) exit
    end do
    if (its == MAXIT+1) call nrerror('too many iterations in gauleg')
```

We use the logical mask `unfinished` to control which vector components are processed inside the `where`. The mask gets updated on each iteration by testing whether any further vector components have converged. When all have converged, we exit the iteration loop. Finally, we check the value of `its` to see whether the maximum allowed number of iterations was exceeded before all components converged.

The logical expression controlling the `where` block (in this case `unfinished`) gets evaluated completely on entry into the `where`, and it is then perfectly fine to modify it inside the block. The modification affects only the *next* execution of the `where`.

On a strictly *serial* machine, there is of course some penalty associated with the above scheme: after a vector component converges, its corresponding component in `unfinished` is redundantly tested on each further iteration, until the slowest-converging component is done. If the number of iterations required does not vary too greatly from component to component, this is a minor, often negligible, penalty. However, one should be on the alert against algorithms whose worst-case convergence could differ from typical convergence by orders of magnitude. For these, one would need to implement a more complicated packing-unpacking scheme. (See discussion in Chapter B6, especially introduction, p. 1083, and notes for `factrl`, p. 1087.)

```
SUBROUTINE gaulag(x,w,alf)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alf
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-13_dp
```

Given `alf`, the parameter α of the Laguerre polynomials, this routine returns arrays `x` and `w` of length N containing the abscissas and weights of the N-point Gauss-Laguerre quadrature formula. The abscissas are returned in ascending order. The parameter `EPS` is the relative precision. Note that internal computations are done in double precision.

```
INTEGER(I4B) :: its,j,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(SP) :: anu
REAL(SP), PARAMETER :: C1=9.084064e-01_sp,C2=5.214976e-02_sp,&
    C3=2.579930e-03_sp,C4=3.986126e-03_sp
REAL(SP), DIMENSION(size(x)) :: rhs,r2,r3,theta
REAL(DP), DIMENSION(size(x)) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION(size(x)) :: unfinished
n=assert_eq(size(x),size(w),'gaulag')
anu=4.0_sp*n+2.0_sp*alf+2.0_sp
rhs=arth(4*n-1,-4,n)*PI/anu
r3=rhs**(1.0_sp/3.0_sp)
r2=r3**2
theta=r3*(C1+r2*(C2+r2*(C3+r2*C4)))
z=anu*cos(theta)**2
unfinished=.true.
do its=1,MAXIT
    where (unfinished)
```

Initial approximations to the roots go into `z`.

Newton's method carried out simultaneously on the roots.

```
        p1=1.0
        p2=0.0
    end where
    do j=1,n                                Loop up the recurrence relation to get the La-
        where (unfinished)                      guerre polynomials evaluated at z.
            p3=p2
            p2=p1
            p1=((2.0_dp*j-1.0_dp+alf-z)*p2-(j-1.0_dp+alf)*p3)/j
        end where
    end do
      p1 now contains the desired Laguerre polynomials. We next compute pp, the derivatives,
      by a standard relation involving also p2, the polynomials of one lower order.
    where (unfinished)
        pp=(n*p1-(n+alf)*p2)/z
        z1=z
        z=z1-p1/pp                          Newton's formula.
        unfinished=(abs(z-z1) > EPS*z)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gaulag')
x=z                                         Store the root and the weight.
w=-exp(gammln(alf+n)-gammln(real(n,sp)))/(pp*n*p2)
END SUBROUTINE gaulag
```

The key difficulty in parallelizing this routine starting from the Fortran 77 version is that the initial guesses for the roots of the Laguerre polynomials were given in terms of previously determined roots. This prevents one from finding all the roots simultaneously. The solution is to come up with a new approximation to the roots that is a simple explicit formula, like the formula we used for the Legendre roots in `gauleg`.

We start with the approximation to $L_n^\alpha(x)$ given in equation (10.15.8) of [1]. We keep only the first term and ask when it is zero. This gives the following prescription for the kth root x_k of $L_n^\alpha(x)$: Solve for θ the equation

$$2\theta - \sin 2\theta = \frac{4n - 4k + 3}{4n + 2\alpha + 2}\pi \tag{B4.1}$$

Since $1 \le k \le n$ and $\alpha > -1$, we can always find a value such that $0 < \theta < \pi/2$. Then the approximation to the root is

$$x_k = (4n + 2\alpha + 2)\cos^2\theta \tag{B4.2}$$

This typically gives 3-digit accuracy, more than enough for the Newton iteration to be able to refine the root. Unfortunately equation (B4.1) is not an explicit formula for θ. (You may recognize it as being of the same form as Kepler's equation in mechanics.) If we call the right-hand side of (B4.1) y, then we can get an explicit formula by working out the power series for $y^{1/3}$ near $\theta = 0$ (using a computer algebra program). Next invert the series to give θ as a function of $y^{1/3}$. Finally, economize the series (see §5.11). The result is the concise approximation

$$\begin{aligned}\theta = {} & 0.9084064y^{1/3} + 5.214976 \times 10^{-2}y + 2.579930 \times 10^{-3}y^{5/3} \\ & + 3.986126 \times 10^{-3}y^{7/3}\end{aligned} \tag{B4.3}$$

```
SUBROUTINE gauher(x,w)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-13_dp,PIM4=0.7511255444649425_dp
```

This routine returns arrays x and w of length N containing the abscissas and weights of the N-point Gauss-Hermite quadrature formula. The abscissas are returned in descending order. Note that internal computations are done in double precision.

Parameters: EPS is the relative precision, PIM4 $= 1/\pi^{1/4}$.

```
INTEGER(I4B) :: its,j,m,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(SP) :: anu
REAL(SP), PARAMETER :: C1=9.084064e-01_sp,C2=5.214976e-02_sp,&
    C3=2.579930e-03_sp,C4=3.986126e-03_sp
REAL(SP), DIMENSION((size(x)+1)/2) :: rhs,r2,r3,theta
REAL(DP), DIMENSION((size(x)+1)/2) :: p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION((size(x)+1)/2) :: unfinished
n=assert_eq(size(x),size(w),'gauher')
m=(n+1)/2                          The roots are symmetric about the origin, so we have to
anu=2.0_sp*n+1.0_sp                    find only half of them.
rhs=arth(3,4,m)*PI/anu
r3=rhs**(1.0_sp/3.0_sp)
r2=r3**2
theta=r3*(C1+r2*(C2+r2*(C3+r2*C4)))
z=sqrt(anu)*cos(theta)             Initial approximations to the roots.
unfinished=.true.
do its=1,MAXIT                     Newton's method carried out simultaneously on the roots.
    where (unfinished)
        p1=PIM4
        p2=0.0
    end where
    do j=1,n                       Loop up the recurrence relation to get the Hermite poly-
        where (unfinished)             nomials evaluated at z.
            p3=p2
            p2=p1
            p1=z*sqrt(2.0_dp/j)*p2-sqrt(real(j-1,dp)/real(j,dp))*p3
        end where
    end do
```

p1 now contains the desired Hermite polynomials. We next compute pp, the derivatives, by the relation (4.5.21) using p2, the polynomials of one lower order.

```
    where (unfinished)
        pp=sqrt(2.0_dp*n)*p2
        z1=z
        z=z1-p1/pp                 Newton's formula.
        unfinished=(abs(z-z1) > EPS)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gauher')
x(1:m)=z                           Store the root
x(n:n-m+1:-1)=-z                   and its symmetric counterpart.
w(1:m)=2.0_dp/pp**2                Compute the weight
w(n:n-m+1:-1)=w(1:m)               and its symmetric counterpart.
END SUBROUTINE gauher
```

Once again we need an explicit approximation for the polynomial roots, this time for $H_n(x)$. We can use the same approximation scheme as for $L_n^\alpha(x)$, since

$$H_{2m}(x) \propto L_m^{-1/2}(x^2), \qquad H_{2m+1}(x) \propto x L_m^{1/2}(x^2) \tag{B4.4}$$

Equations (B4.1) and (B4.2) become

$$\begin{aligned} 2\theta - \sin 2\theta &= \frac{4k-1}{2n+1}\pi \\ x_k &= \sqrt{2n+1}\cos\theta \end{aligned} \tag{B4.5}$$

Here $k = 1, 2, \ldots, m$ where $m = [(n+1)/2]$, and $k = 1$ is the largest root. The negative roots follow from symmetry. The root at $x = 0$ for odd n is included in this approximation.

```
SUBROUTINE gaujac(x,w,alf,bet)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alf,bet
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
REAL(DP), PARAMETER :: EPS=3.0e-14_dp
```

Given alf and bet, the parameters α and β of the Jacobi polynomials, this routine returns arrays x and w of length N containing the abscissas and weights of the N-point Gauss-Jacobi quadrature formula. The abscissas are returned in descending order. The parameter EPS is the relative precision. Note that internal computations are done in double precision.

```
INTEGER(I4B) :: its,j,n
INTEGER(I4B), PARAMETER :: MAXIT=10
REAL(DP) :: alfbet,a,c,temp
REAL(DP), DIMENSION(size(x)) :: b,p1,p2,p3,pp,z,z1
LOGICAL(LGT), DIMENSION(size(x)) :: unfinished
n=assert_eq(size(x),size(w),'gaujac')
alfbet=alf+bet                          Initial approximations to the roots go into z.
z=cos(PI*(arth(1,1,n)-0.25_dp+0.5_dp*alf)/(n+0.5_dp*(alfbet+1.0_dp)))
unfinished=.true.
do its=1,MAXIT                          Newton's method carried out simultaneously on the roots.
    temp=2.0_dp+alfbet
    where (unfinished)                  Start the recurrence with P_0 and P_1 to avoid a division
        p1=(alf-bet+temp*z)/2.0_dp          by zero when α + β = 0 or −1.
        p2=1.0
    end where
    do j=2,n                            Loop up the recurrence relation to get the Jacobi poly-
        a=2*j*(j+alfbet)*temp               nomials evaluated at z.
        temp=temp+2.0_dp
        c=2.0_dp*(j-1.0_dp+alf)*(j-1.0_dp+bet)*temp
        where (unfinished)
            p3=p2
            p2=p1
            b=(temp-1.0_dp)*(alf*alf-bet*bet+temp*&
                (temp-2.0_dp)*z)
            p1=(b*p2-c*p3)/a
        end where
    end do
```

p1 now contains the desired Jacobi polynomials. We next compute pp, the derivatives, by a standard relation involving also p2, the polynomials of one lower order.

```
    where (unfinished)
        pp=(n*(alf-bet-temp*z)*p1+2.0_dp*(n+alf)*&
            (n+bet)*p2)/(temp*(1.0_dp-z*z))
        z1=z
        z=z1-p1/pp                      Newton's formula.
        unfinished=(abs(z-z1) > EPS)
    end where
    if (.not. any(unfinished)) exit
end do
if (its == MAXIT+1) call nrerror('too many iterations in gaujac')
x=z                                     Store the root and the weight.
```

```
w=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.0_sp)-&
    gammln(n+alf+bet+1.0_sp))*temp*2.0_sp**alfbet/(pp*p2)
END SUBROUTINE gaujac
```

Now we need an explicit approximation for the roots of the Jacobi polynomials $P_n^{(\alpha,\beta)}(x)$. We start with the asymptotic expansion (10.14.10) of [1]. Setting this to zero gives the formula

$$x = \cos\left[\frac{k - 1/4 + \alpha/2}{n + (\alpha+\beta+1)/2}\pi\right] \tag{B4.6}$$

This is better than the formula (22.16.1) in [2], especially at small and moderate n.

⋆ ⋆ ⋆

```
SUBROUTINE gaucof(a,b,amu0,x,w)
USE nrtype; USE nrutil, ONLY : assert_eq,unit_matrix
USE nr, ONLY : eigsrt,tqli
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
REAL(SP), INTENT(IN) :: amu0
REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
```

Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi matrix. On input, a and b of length N are the coefficients of the recurrence relation for the set of monic orthogonal polynomials. The quantity $\mu_0 \equiv \int_a^b W(x)\,dx$ is input as amu0. The abscissas are returned in descending order in array x of length N, with the corresponding weights in w, also of length N. The arrays a and b are modified. Execution can be speeded up by modifying tqli and eigsrt to compute only the first component of each eigenvector.

```
REAL(SP), DIMENSION(size(a),size(a)) :: z
INTEGER(I4B) :: n
n=assert_eq(size(a),size(b),size(x),size(w),'gaucof')
b(2:n)=sqrt(b(2:n))          Set up superdiagonal of Jacobi matrix.
call unit_matrix(z)          Set up identity matrix for tqli to compute eigenvectors.
call tqli(a,b,z)
call eigsrt(a,z)             Sort eigenvalues into descending order.
x=a
w=amu0*z(1,:)**2             Equation (4.5.12).
END SUBROUTINE gaucof
```

⋆ ⋆ ⋆

```
SUBROUTINE orthog(anu,alpha,beta,a,b)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: anu,alpha,beta
REAL(SP), DIMENSION(:), INTENT(OUT) :: a,b
```

Computes the coefficients a_j and b_j, $j = 0, \ldots N-1$, of the recurrence relation for monic orthogonal polynomials with weight function $W(x)$ by Wheeler's algorithm. On input, alpha and beta contain the $2N-1$ coefficients α_j and β_j, $j = 0, \ldots 2N-2$, of the recurrence

relation for the chosen basis of orthogonal polynomials. The $2N$ modified moments ν_j are input in anu for $j = 0, \ldots 2N-1$. The first N coefficients are returned in a and b.

```
INTEGER(I4B) :: k,n,ndum
REAL(SP), DIMENSION(2*size(a)+1,2*size(a)+1) :: sig
n=assert_eq(size(a),size(b),'orthog: n')
ndum=assert_eq(2*n,size(alpha)+1,size(anu),size(beta)+1,'orthog: ndum')
sig(1,3:2*n)=0.0                Initialization, Equation (4.5.33).
sig(2,2:2*n+1)=anu(1:2*n)
a(1)=alpha(1)+anu(2)/anu(1)
b(1)=0.0
do k=3,n+1                      Equation (4.5.34).
    sig(k,k:2*n-k+3)=sig(k-1,k+1:2*n-k+4)+(alpha(k-1:2*n-k+2) &
        -a(k-2))*sig(k-1,k:2*n-k+3)-b(k-2)*sig(k-2,k:2*n-k+3) &
        +beta(k-1:2*n-k+2)*sig(k-1,k-1:2*n-k+2)
    a(k-1)=alpha(k-1)+sig(k,k+1)/sig(k,k)-sig(k-1,k)/sig(k-1,k-1)
    b(k-1)=sig(k,k)/sig(k-1,k-1)
end do
END SUBROUTINE orthog
```

⋆ ⋆ ⋆

f90 As discussed in Volume 1, multidimensional quadrature can be performed by calling a one-dimensional quadrature routine along each dimension. If the same routine is used for all such calls, then the calls are recursive. The file quad3d.f90 contains two modules, quad3d_qgaus_mod and quad3d_qromb_mod. In the first, the basic one-dimensional quadrature routine is a 10-point Gaussian quadrature routine called qgaus and three-dimensional quadrature is performed by calling quad3d_qgaus. In the second, the basic one-dimensional routine is qromb of §4.3 and the three-dimensional routine is quad3d_qromb. The Gaussian quadrature is simpler but its accuracy is not controllable. The Romberg integration lets you specify an accuracy, but is apt to be very slow if you try for too much accuracy. The only difference between the stand-alone version of trapzd and the version included here is that we have to add the keyword RECURSIVE. The only changes from the stand-alone version of qromb are: We have to add RECURSIVE; we remove trapzd from the list of routines in USE nr; we increase EPS to 3×10^{-6}. Even this value could be too ambitious for difficult functions. You may want to set JMAX to a smaller value than 20 to avoid burning up a lot of computer time. Some people advocate using a smaller EPS on the inner quadrature (over z in our routine) than on the outer quadratures (over x or y). That strategy would require separate copies of qromb.

```
MODULE quad3d_qgaus_mod
USE nrtype
PRIVATE                         Hide all names from the outside,
PUBLIC quad3d_qgaus             except quad3d itself.
REAL(SP) :: xsav,ysav
INTERFACE                       User-supplied functions.
    FUNCTION func(x,y,z)        The three-dimensional function to be integrated.
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), INTENT(IN) :: z
    REAL(SP), DIMENSION(size(z)) :: func
    END FUNCTION func

    FUNCTION y1(x)
    USE nrtype
```

```
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y1
    END FUNCTION y1

    FUNCTION y2(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y2
    END FUNCTION y2

    FUNCTION z1(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z1
    END FUNCTION z1

    FUNCTION z2(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z2
    END FUNCTION z2
END INTERFACE
```

The routine quad3d_qgaus returns as ss the integral of a user-supplied function func over a three-dimensional region specified by the limits x1, x2, and by the user-supplied functions y1, y2, z1, and z2, as defined in (4.6.2). Integration is performed by calling qgaus recursively.

```
CONTAINS

FUNCTION h(x)                          This is H of eq. (4.6.5).
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: h
INTEGER(I4B) :: i
do i=1,size(x)
    xsav=x(i)
    h(i)=qgaus(g,y1(xsav),y2(xsav))
end do
END FUNCTION h

FUNCTION g(y)                          This is G of eq. (4.6.4).
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(size(y)) :: g
INTEGER(I4B) :: j
do j=1,size(y)
    ysav=y(j)
    g(j)=qgaus(f,z1(xsav,ysav),z2(xsav,ysav))
end do
END FUNCTION g

FUNCTION f(z)                          The integrand f(x, y, z) evaluated at fixed x and y.
REAL(SP), DIMENSION(:), INTENT(IN) :: z
REAL(SP), DIMENSION(size(z)) :: f
f=func(xsav,ysav,z)
END FUNCTION f

RECURSIVE FUNCTION qgaus(func,a,b)
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qgaus
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
REAL(SP) :: xm,xr
REAL(SP), DIMENSION(5) :: dx, w = (/ 0.2955242247_sp,0.2692667193_sp,&
    0.2190863625_sp,0.1494513491_sp,0.0666713443_sp /),&
    x = (/ 0.1488743389_sp,0.4333953941_sp,0.6794095682_sp,&
```

```
    0.8650633666_sp,0.9739065285_sp /)
xm=0.5_sp*(b+a)
xr=0.5_sp*(b-a)
dx(:)=xr*x(:)
qgaus=xr*sum(w(:)*(func(xm+dx)+func(xm-dx)))
END FUNCTION qgaus

SUBROUTINE quad3d_qgaus(x1,x2,ss)
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: ss
ss=qgaus(h,x1,x2)
END SUBROUTINE quad3d_qgaus
END MODULE quad3d_qgaus_mod
```

`PRIVATE...PUBLIC quad3d_qgaus` By default, all module entities are accessible by a routine that uses the module (unless we restrict the `USE` statement with `ONLY`). In this module, the user needs access only to the routine `quad3d_qgaus`; the variables `xsav`, `ysav` and the procedures `f`, `g`, `h`, and `qgaus` are purely internal. It is good programming practice to prevent duplicate name conflicts or data overwriting by limiting access to only the desired entities. Here the `PRIVATE` statement with no variable names resets the default from `PUBLIC`. Then we include in the `PUBLIC` statement only the function name we want to be accessible.

`REAL(SP) :: xsav,ysav` In Fortran 90, we generally avoid declaring global variables in `COMMON` blocks. Instead, we give them complete specifications in a module. A deficiency of Fortran 90 is that it does not allow pointers to functions. So here we have to use the fixed-name function `func` for the function to be integrated over. If we could have a pointer to a function as a global variable, then we would just set the pointer to point to the user function (of any name) in the calling program. Similarly the functions `y1`, `y2`, `z1`, and `z2` could also have any name.

`CONTAINS` Here follow the internal subprograms `f`, `g`, `h`, `qgaus`, and `quad3d_qgaus`. Note that such internal subprograms are all "visible" to each other, i.e., their interfaces are mutually explicit, and do not require `INTERFACE` statements.

`RECURSIVE SUBROUTINE qgaus(func,a,b,ss)` The `RECURSIVE` keyword is required for the compiler to process correctly any procedure that is invoked again in its body before the return from the first call has been completed. While some compilers may let you get away without explicitly informing them that a routine is recursive, don't count on it!

```
MODULE quad3d_qromb_mod
    Alternative to quad3d_qgaus_mod that uses qromb to perform each one-dimensional in-
    tegration.
USE nrtype
PRIVATE
PUBLIC quad3d_qromb
REAL(SP) :: xsav,ysav
INTERFACE
    FUNCTION func(x,y,z)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), INTENT(IN) :: z
    REAL(SP), DIMENSION(size(z)) :: func
    END FUNCTION func

    FUNCTION y1(x)
```

```
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y1
    END FUNCTION y1

    FUNCTION y2(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: y2
    END FUNCTION y2

    FUNCTION z1(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z1
    END FUNCTION z1

    FUNCTION z2(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: z2
    END FUNCTION z2
END INTERFACE
CONTAINS

FUNCTION h(x)
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: h
INTEGER(I4B) :: i
do i=1,size(x)
    xsav=x(i)
    h(i)=qromb(g,y1(xsav),y2(xsav))
end do
END FUNCTION h

FUNCTION g(y)
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(size(y)) :: g
INTEGER(I4B) :: j
do j=1,size(y)
    ysav=y(j)
    g(j)=qromb(f,z1(xsav,ysav),z2(xsav,ysav))
end do
END FUNCTION g

FUNCTION f(z)
REAL(SP), DIMENSION(:), INTENT(IN) :: z
REAL(SP), DIMENSION(size(z)) :: f
f=func(xsav,ysav,z)
END FUNCTION f

RECURSIVE FUNCTION qromb(func,a,b)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : polint
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP) :: qromb
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: JMAX=20,JMAXP=JMAX+1,K=5,KM=K-1
REAL(SP), PARAMETER :: EPS=3.0e-6_sp
REAL(SP), DIMENSION(JMAXP) :: h,s
REAL(SP) :: dqromb
```

```
INTEGER(I4B) :: j
h(1)=1.0
do j=1,JMAX
    call trapzd(func,a,b,s(j),j)
    if (j >= K) then
        call polint(h(j-KM:j),s(j-KM:j),0.0_sp,qromb,dqromb)
        if (abs(dqromb) <= EPS*abs(qromb)) RETURN
    end if
    s(j+1)=s(j)
    h(j+1)=0.25_sp*h(j)
end do
call nrerror('qromb: too many steps')
END FUNCTION qromb

RECURSIVE SUBROUTINE trapzd(func,a,b,s,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), INTENT(INOUT) :: s
INTEGER(I4B), INTENT(IN) :: n
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
REAL(SP) :: del,fsum
INTEGER(I4B) :: it
if (n == 1) then
    s=0.5_sp*(b-a)*sum(func( (/ a,b /) ))
else
    it=2**(n-2)
    del=(b-a)/it
    fsum=sum(func(arth(a+0.5_sp*del,del,it)))
    s=0.5_sp*(s+del*fsum)
end if
END SUBROUTINE trapzd

SUBROUTINE quad3d_qromb(x1,x2,ss)
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), INTENT(OUT) :: ss
ss=qromb(h,x1,x2)
END SUBROUTINE quad3d_qromb
END MODULE quad3d_qromb_mod
```

`MODULE quad3d_qromb_mod` The only difference between this module and the previous one is that all calls to `qgaus` are replaced by calls to `qromb` and that the routine `qgaus` is replaced by `qromb` and `trapzd`.

CITED REFERENCES AND FURTHER READING:

Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendental Functions*, Volume II (New York: McGraw-Hill). [1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [2]

```
SUBROUTINE eulsum(sum,term,jterm)
USE nrtype; USE nrutil, ONLY : poly_term,reallocate
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: sum
REAL(SP), INTENT(IN) :: term
INTEGER(I4B), INTENT(IN) :: jterm
```

Incorporates into sum the jterm'th term, with value term, of an alternating series. sum is input as the previous partial sum, and is output as the new partial sum. The first call to this routine, with the first term in the series, should be with jterm=1. On the second call, term should be set to the second term of the series, with sign opposite to that of the first call, and jterm should be 2. And so on.

```
REAL(SP), DIMENSION(:), POINTER, SAVE :: wksp
INTEGER(I4B), SAVE :: nterm                    Number of saved differences in wksp.
LOGICAL(LGT), SAVE :: init=.true.
if (init) then                                 Initialize.
    init=.false.
    nullify(wksp)
end if
if (jterm == 1) then
    nterm=1
    wksp=>reallocate(wksp,100)
    wksp(1)=term
    sum=0.5_sp*term                            Return first estimate.
else
    if (nterm+1 > size(wksp)) wksp=>reallocate(wksp,2*size(wksp))
    wksp(2:nterm+1)=0.5_sp*wksp(1:nterm)       Update saved quantities by van Wijn-
    wksp(1)=term                                   gaarden's algorithm.
    wksp(1:nterm+1)=poly_term(wksp(1:nterm+1),0.5_sp)
    if (abs(wksp(nterm+1)) <= abs(wksp(nterm))) then    Favorable to increase p,
        sum=sum+0.5_sp*wksp(nterm+1)
        nterm=nterm+1                          and the table becomes longer.
    else                                       Favorable to increase n,
        sum=sum+wksp(nterm+1)                  the table doesn't become longer.
    end if
end if
END SUBROUTINE eulsum
```

f90 This routine uses the function reallocate in nrutil to define a temporary workspace and then, if necessary, enlarge the workspace without destroying the earlier contents. The pointer wksp is declared with the SAVE attribute. Since Fortran 90 pointers are born "in limbo," we cannot immediately test whether they are associated or not. Hence the code if (init)...nullify(wksp). Then the line wksp=>reallocate(wksp,100) allocates an array of length 100 and points wksp to it. On subsequent calls to eulsum, if nterm ever gets bigger than the size of wksp, the call to reallocate doubles the size of wksp and copies the old contents into the new storage.

You could achieve the same effect as the code if (init)...nullify(wksp)... wksp=>reallocate(wksp,100) with a simple allocate(wksp,100). You would then use

reallocate only for increasing the storage if necessary. Don't! The advantage of the above scheme becomes clear if you consider what happens if eulsum is invoked *twice* by the calling program to evaluate two different sums. On the second invocation, when jterm $= 1$ again, you would be allocating an already allocated pointer. This does not generate an error — it simply leaves the original target inaccessible. Using reallocate instead not only allocates a new array of length 100, but also detects that wksp had already been associated. It dutifully (and wastefully) copies the first 100 elements of the old wksp into the new storage, and, more importantly, deallocates the old wksp, reclaiming its storage. While only two invocations of eulsum without intervening deallocation of memory would not cause a problem, many such invocations might well. We believe that, as a general rule, the potential for catastrophe from reckless use of allocate is great enough that you should *always* deallocate whenever storage is no longer required.

The unnecessary copying of 100 elements when eulsum is invoked a second time could be avoided by making init an argument. It hardly seems worth it to us.

For Fortran 90 neophytes, note that unlike in C you have to do nothing special to get the contents of the storage a pointer is addressing. The compiler figures out from the context whether you mean the contents, such as wksp(1:nterm), or the address, such as both occurrences of wksp in wksp=>reallocate(wksp,100).

wksp(1:nterm+1)=poly_term(wksp(1:nterm+1),0.5_sp) The poly_term function in nrutil tabulates the partial sums of a polynomial, or, equivalently, performs the synthetic division of a polynomial by a monomial.

Small-scale parallelism in eulsum is achieved straightforwardly by the use of vector constructions and poly_term (which parallelizes recursively). The routine is not written to take advantage of data parallelism in the (infrequent) case of wanting to sum many different series simultaneously; nor, since wksp is a SAVEd variable, can it be used in many simultaneous instances on a MIMD machine. (You can easily recode these generalizations if you need them.)

⋆ ⋆ ⋆

```
SUBROUTINE ddpoly(c,x,pd)
USE nrtype; USE nrutil, ONLY : arth,cumprod,poly_term
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(:), INTENT(OUT) :: pd
```

Given the coefficients of a polynomial of degree $N_c - 1$ as an array c(1:N_c) with c(1) being the constant term, and given a value x, this routine returns the polynomial evaluated at x as pd(1) and $N_d - 1$ derivatives as pd(2:N_d).

```
INTEGER(I4B) :: i,nc,nd
REAL(SP), DIMENSION(size(pd)) :: fac
REAL(SP), DIMENSION(size(c)) :: d
nc=size(c)
nd=size(pd)
d(nc:1:-1)=poly_term(c(nc:1:-1),x)
do i=2,min(nd,nc)
    d(nc:i:-1)=poly_term(d(nc:i:-1),x)
end do
pd=d(1:nd)
fac=cumprod(arth(1.0_sp,1.0_sp,nd))
pd(3:nd)=fac(2:nd-1)*pd(3:nd)
END SUBROUTINE ddpoly
```

After the first derivative, factorial constants come in.

f90 `d(nc:1:-1)=poly_term(c(nc:1:-1),x)` The `poly_term` function in `nrutil` tabulates the partial sums of a polynomial, or, equivalently, performs synthetic division. See §22.3 for a discussion of why `ddpoly` is coded this way.

`fac=cumprod(arth(1.0_sp,1.0_sp,nd))` Here the function `arth` from `nrutil` generates the sequence $1, 2, 3, \ldots$. The function `cumprod` then tabulates the cumulative products, thus making a table of factorials.

Notice that `ddpoly` doesn't need an argument to pass N_d, the number of output terms desired by the user: It gets that information from the length of the array `pd` that the user provides for it to fill. It is a minor curiosity that `pd`, declared as `INTENT(OUT)`, can thus be used, on the sly, to pass some `INTENT(IN)` information. (A Fortran 90 brain teaser could be: A subroutine with only `INTENT(OUT)` arguments can be called to print any specified integer. How is this done?)

```
SUBROUTINE poldiv(u,v,q,r)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: u,v
REAL(SP), DIMENSION(:), INTENT(OUT) :: q,r
```

Given the N coefficients of a polynomial in `u`, and the N_v coefficients of another polynomial in `v`, divide the polynomial `u` by the polynomial `v` ("`u`"/"`v`") giving a quotient polynomial whose coefficients are returned in `q`, and a remainder polynomial whose coefficients are returned in `r`. The arrays `q` and `r` are of length N, but only the first $N - N_v + 1$ elements of `q` and the first $N_v - 1$ elements of `r` are used. The remaining elements are returned as zero.

```
INTEGER(I4B) :: i,n,nv
n=assert_eq(size(u),size(q),size(r),'poldiv')
nv=size(v)
r(:)=u(:)
q(:)=0.0
do i=n-nv,0,-1
    q(i+1)=r(nv+i)/v(nv)
    r(i+1:nv+i-1)=r(i+1:nv+i-1)-q(i+1)*v(1:nv-1)
end do
r(nv:n)=0.0
END SUBROUTINE poldiv
```

$\star \quad \star \quad \star$

```
FUNCTION ratval_s(x,cof,mm,kk)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(DP), INTENT(IN) :: x          Note precision! Change to REAL(SP) if desired.
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP) :: ratval_s
```

Given `mm`, `kk`, and `cof(1:mm+kk+1)`, evaluate and return the rational function $(\texttt{cof(1)} + \texttt{cof(2)}\,x + \cdots + \texttt{cof(mm+1)}\,x^{\texttt{mm}})/(1 + \texttt{cof(mm+2)}\,x + \cdots + \texttt{cof(mm+kk+1)}\,x^{\texttt{kk}})$.

```
ratval_s=poly(x,cof(1:mm+1))/(1.0_dp+x*poly(x,cof(mm+2:mm+kk+1)))
END FUNCTION ratval_s
```

f90 This simple routine uses the function `poly` from `nrutil` to evaluate the numerator and denominator polynomials. Single- and double-precision versions, `ratval_s` and `ratval_v`, are overloaded onto the name `ratval` when the module `nr` is used.

```
FUNCTION ratval_v(x,cof,mm,kk)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
REAL(DP), DIMENSION(size(x)) :: ratval_v
ratval_v=poly(x,cof(1:mm+1))/(1.0_dp+x*poly(x,cof(mm+2:mm+kk+1)))
END FUNCTION ratval_v
```

⋆ ⋆ ⋆

The routines `recur1` and `recur2` are new in this volume, and do not have Fortran 77 counterparts. First- and second-order linear recurrences are implemented as trivial do-loops on strictly serial machines. On parallel machines, however, they pose different, and quite interesting, programming challenges. Since many calculations can be decomposed into recurrences, it is useful to have general, parallelizable routines available. The algorithms behind `recur1` and `recur2` are discussed in §22.2.

```
RECURSIVE FUNCTION recur1(a,b) RESULT(u)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
REAL(SP), DIMENSION(size(a)) :: u
INTEGER(I4B), PARAMETER :: NPAR_RECUR1=8
    Given vectors a of size n and b of size n − 1, returns a vector u that satisfies the first
    order linear recurrence u_1 = a_1, u_j = a_j + b_{j−1} u_{j−1}, for j = 2, ..., n. Parallelization is
    via a recursive evaluation.
INTEGER(I4B) :: n,j
n=assert_eq(size(a),size(b)+1,'recur1')
u(1)=a(1)
if (n < NPAR_RECUR1) then          Do short vectors as a loop.
    do j=2,n
        u(j)=a(j)+b(j-1)*u(j-1)
    end do
else
      Otherwise, combine coefficients and recurse on the even components, then evaluate all
      the odd components in parallel.
    u(2:n:2)=recur1(a(2:n:2)+a(1:n-1:2)*b(1:n-1:2), &
           b(3:n-1:2)*b(2:n-2:2))
    u(3:n:2)=a(3:n:2)+b(2:n-1:2)*u(2:n-1:2)
end if
END FUNCTION recur1
```

f90 `RECURSIVE FUNCTION recur1(a,b) RESULT(u)` When a recursive function invokes itself only indirectly through a sequence of function calls, then the function name can be used for the result just as in a nonrecursive function. When the function invokes itself directly, however, as in `recur1`, then another name must be used for the result. If you are hazy on the syntax for `RESULT`, see the discussion of recursion in §21.5.

⋆ ⋆ ⋆

```
FUNCTION recur2(a,b,c)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c
REAL(SP), DIMENSION(size(a)) :: recur2
```

Given vectors a of size n and b and c of size $n-2$, returns a vector u that satisfies the second order linear recurrence $u_1 = a_1$, $u_2 = a_2$, $u_j = a_j + b_{j-2}u_{j-1} + c_{j-2}u_{j-2}$, for $j = 3, \ldots, n$. Parallelization is via conversion to a first order recurrence for a two-dimensional vector.

```
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(a)-1) :: a1,a2,u1,u2
REAL(SP), DIMENSION(size(a)-2) :: b11,b12,b21,b22
n=assert_eq(size(a),size(b)+2,size(c)+2,'recur2')
a1(1)=a(1)                          Set up vector a.
a2(1)=a(2)
a1(2:n-1)=0.0
a2(2:n-1)=a(3:n)
b11(1:n-2)=0.0                      Set up matrix b.
b12(1:n-2)=1.0
b21(1:n-2)=c(1:n-2)
b22(1:n-2)=b(1:n-2)
call recur1_v(a1,a2,b11,b12,b21,b22,u1,u2)
recur2(1:n-1)=u1(1:n-1)
recur2(n)=u2(n-1)
CONTAINS
RECURSIVE SUBROUTINE recur1_v(a1,a2,b11,b12,b21,b22,u1,u2)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a1,a2,b11,b12,b21,b22
REAL(SP), DIMENSION(:), INTENT(OUT) :: u1,u2
INTEGER(I4B), PARAMETER :: NPAR_RECUR2=8
```

Used by recur2 to evaluate first order vector recurrence. Routine is a two-dimensional vector version of recur1, with matrix multiplication replacing scalar multiplication.

```
INTEGER(I4B) :: n,j,nn,nn1
REAL(SP), DIMENSION(size(a1)/2) :: aa1,aa2
REAL(SP), DIMENSION(size(a1)/2-1) :: bb11,bb12,bb21,bb22
n=assert_eq((/size(a1),size(a2),size(b11)+1,size(b12)+1,size(b21)+1,&
    size(b22)+1,size(u1),size(u2)/),'recur1_v')
u1(1)=a1(1)
u2(1)=a2(1)
if (n < NPAR_RECUR2) then           Do short vectors as a loop.
    do j=2,n
        u1(j)=a1(j)+b11(j-1)*u1(j-1)+b12(j-1)*u2(j-1)
        u2(j)=a2(j)+b21(j-1)*u1(j-1)+b22(j-1)*u2(j-1)
    end do
else
```

Otherwise, combine coefficients and recurse on the even components, then evaluate all the odd components in parallel.

```
    nn=n/2
    nn1=nn-1
    aa1(1:nn)=a1(2:n:2)+b11(1:n-1:2)*a1(1:n-1:2)+&
        b12(1:n-1:2)*a2(1:n-1:2)
    aa2(1:nn)=a2(2:n:2)+b21(1:n-1:2)*a1(1:n-1:2)+&
            b22(1:n-1:2)*a2(1:n-1:2)
    bb11(1:nn1)=b11(3:n-1:2)*b11(2:n-2:2)+&
            b12(3:n-1:2)*b21(2:n-2:2)
    bb12(1:nn1)=b11(3:n-1:2)*b12(2:n-2:2)+&
            b12(3:n-1:2)*b22(2:n-2:2)
    bb21(1:nn1)=b21(3:n-1:2)*b11(2:n-2:2)+&
            b22(3:n-1:2)*b21(2:n-2:2)
    bb22(1:nn1)=b21(3:n-1:2)*b12(2:n-2:2)+&
            b22(3:n-1:2)*b22(2:n-2:2)
    call recur1_v(aa1,aa2,bb11,bb12,bb21,bb22,u1(2:n:2),u2(2:n:2))
    u1(3:n:2)=a1(3:n:2)+b11(2:n-1:2)*u1(2:n-1:2)+&
```

```
            b12(2:n-1:2)*u2(2:n-1:2)
    u2(3:n:2)=a2(3:n:2)+b21(2:n-1:2)*u1(2:n-1:2)+&
            b22(2:n-1:2)*u2(2:n-1:2)
end if
END SUBROUTINE recur1_v
END FUNCTION recur2
```

⋆ ⋆ ⋆

```
FUNCTION dfridr(func,x,h,err)
USE nrtype; USE nrutil, ONLY : assert,geop,iminloc
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,h
REAL(SP), INTENT(OUT) :: err
REAL(SP) :: dfridr
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B),PARAMETER :: NTAB=10
REAL(SP), PARAMETER :: CON=1.4_sp,CON2=CON*CON,BIG=huge(x),SAFE=2.0
```

Returns the derivative of a function func at a point x by Ridders' method of polynomial extrapolation. The value h is input as an estimated initial stepsize; it need not be small, but rather should be an increment in x over which func changes *substantially*. An estimate of the error in the derivative is returned as err.

Parameters: Stepsize is decreased by CON at each iteration. Max size of tableau is set by NTAB. Return when error is SAFE worse than the best so far.

```
INTEGER(I4B) :: ierrmin,i,j
REAL(SP) :: hh
REAL(SP), DIMENSION(NTAB-1) :: errt,fac
REAL(SP), DIMENSION(NTAB,NTAB) :: a
call assert(h /= 0.0, 'dfridr arg')
hh=h
a(1,1)=(func(x+hh)-func(x-hh))/(2.0_sp*hh)
err=BIG
fac(1:NTAB-1)=geop(CON2,CON2,NTAB-1)
do i=2,NTAB                          Successive columns in the Neville tableau will go to smaller
    hh=hh/CON                            stepsizes and higher orders of extrapolation.
    a(1,i)=(func(x+hh)-func(x-hh))/(2.0_sp*hh)        Try new, smaller stepsize.
    do j=2,i
          Compute extrapolations of various orders, requiring no new function evaluations.
        a(j,i)=(a(j-1,i)*fac(j-1)-a(j-1,i-1))/(fac(j-1)-1.0_sp)
    end do
    errt(1:i-1)=max(abs(a(2:i,i)-a(1:i-1,i)),abs(a(2:i,i)-a(1:i-1,i-1)))
      The error strategy is to compare each new extrapolation to one order lower, both at the
      present stepsize and the previous one.
    ierrmin=iminloc(errt(1:i-1))
    if (errt(ierrmin) <= err) then          If error is decreased, save the improved an-
        err=errt(ierrmin)                       swer.
        dfridr=a(1+ierrmin,i)
    end if
    if (abs(a(i,i)-a(i-1,i-1)) >= SAFE*err) RETURN
      If higher order is worse by a significant factor SAFE, then quit early.
end do
END FUNCTION dfridr
```

f90 `ierrmin=iminloc(errt(1:i-1))` The function `iminloc` in `nrutil` is useful when you need to know the index of the smallest element in an array.

⋆ ⋆ ⋆

```
FUNCTION chebft(a,b,n,func)
USE nrtype; USE nrutil, ONLY : arth,outerprod
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: chebft
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
```

Chebyshev fit: Given a function `func`, lower and upper limits of the interval [a,b], and a maximum degree `n`, this routine computes the `n` coefficients c_k such that $\texttt{func}(x) \approx [\sum_{k=1}^{n} c_k T_{k-1}(y)] - c_1/2$, where y and x are related by (5.8.10). This routine is to be used with moderately large `n` (e.g., 30 or 50), the array of `c`'s subsequently to be truncated at the smaller value m such that c_{m+1} and subsequent elements are negligible.

```
REAL(DP) :: bma,bpa
REAL(DP), DIMENSION(n) :: theta
bma=0.5_dp*(b-a)
bpa=0.5_dp*(b+a)
theta(:)=PI_D*arth(0.5_dp,1.0_dp,n)/n
chebft(:)=matmul(cos(outerprod(arth(0.0_dp,1.0_dp,n),theta)), &
    func(real(cos(theta)*bma+bpa,sp)))*2.0_dp/n
```

We evaluate the function at the `n` points required by (5.8.7). We accumulate the sum in double precision for safety.

```
END FUNCTION chebft
```

f90 `chebft(:)=matmul(...)` Here again Fortran 90 produces a very concise parallelizable formulation that requires some effort to decode. Equation (5.8.7) is a product of the matrix of cosines, where the rows are indexed by j and the columns by k, with the vector of function values indexed by k. We use the `outerprod` function in `nrutil` to form the matrix of arguments for the cosine, and rely on the element-by-element application of `cos` to produce the matrix of cosines. `matmul` then takes care of the matrix product. A subtlety is that, while the calculation is being done in double precision to minimize roundoff, the function is assumed to be supplied in single precision. Thus `real(...,sp)` is used to convert the double precision argument to single precision.

```
FUNCTION chebev_s(a,b,c,x)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,x
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP) :: chebev_s
```

Chebyshev evaluation: All arguments are input. `c` is an array of length M of Chebyshev coefficients, the first M elements of `c` output from `chebft` (which must have been called

with the same a and b). The Chebyshev polynomial $\sum_{k=1}^{M} c_k T_{k-1}(y) - c_1/2$ is evaluated at a point $y = [x-(b+a)/2]/[(b-a)/2]$, and the result is returned as the function value.

```
INTEGER(I4B) :: j,m
REAL(SP) :: d,dd,sv,y,y2
if ((x-a)*(x-b) > 0.0) call nrerror('x not in range in chebev_s')
m=size(c)
d=0.0
dd=0.0
y=(2.0_sp*x-a-b)/(b-a)              Change of variable.
y2=2.0_sp*y
do j=m,2,-1                         Clenshaw's recurrence.
    sv=d
    d=y2*d-dd+c(j)
    dd=sv
end do
chebev_s=y*d-dd+0.5_sp*c(1)         Last step is different.
END FUNCTION chebev_s
```

```
FUNCTION chebev_v(a,b,c,x)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c,x
REAL(SP), DIMENSION(size(x)) :: chebev_v
INTEGER(I4B) :: j,m
REAL(SP), DIMENSION(size(x)) :: d,dd,sv,y,y2
if (any((x-a)*(x-b) > 0.0)) call nrerror('x not in range in chebev_v')
m=size(c)
d=0.0
dd=0.0
y=(2.0_sp*x-a-b)/(b-a)
y2=2.0_sp*y
do j=m,2,-1
    sv=d
    d=y2*d-dd+c(j)
    dd=sv
end do
chebev_v=y*d-dd+0.5_sp*c(1)
END FUNCTION chebev_v
```

f90 The name chebev is overloaded with scalar and vector versions. chebev_v is essentially identical to chebev_s except for the declarations of the variables. Fortran 90 does the appropriate scalar or vector arithmetic in the body of the routine, depending on the type of the variables.

⋆ ⋆ ⋆

```
FUNCTION chder(a,b,c)
USE nrtype; USE nrutil, ONLY : arth,cumsum
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chder
```

This routine returns an array of length N containing the Chebyshev coefficients of the derivative of the function whose coefficients are in the array c. Input are a, b, c, as output

from routine chebft §5.8. The desired degree of approximation N is equal to the length of c supplied.

```
INTEGER(I4B) :: n
REAL(SP) :: con
REAL(SP), DIMENSION(size(c)) :: temp
n=size(c)
temp(1)=0.0
temp(2:n)=2.0_sp*arth(n-1,-1,n-1)*c(n:2:-1)
chder(n:1:-2)=cumsum(temp(1:n:2))          Equation (5.9.2).
chder(n-1:1:-2)=cumsum(temp(2:n:2))
con=2.0_sp/(b-a)
chder=chder*con                            Normalize to the interval b-a.
END FUNCTION chder
```

```
FUNCTION chint(a,b,c)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chint
```

This routine returns an array of length N containing the Chebyshev coefficients of the integral of the function whose coefficients are in the array c. Input are a,b,c, as output from routine chebft §5.8. The desired degree of approximation N is equal to the length of c supplied. The constant of integration is set so that the integral vanishes at a.

```
INTEGER(I4B) :: n
REAL(SP) :: con
n=size(c)
con=0.25_sp*(b-a)                          Factor that normalizes to the interval b-a.
chint(2:n-1)=con*(c(1:n-2)-c(3:n))/arth(1,1,n-2)   Equation (5.9.1).
chint(n)=con*c(n-1)/(n-1)                  Special case of (5.9.1) for n.
chint(1)=2.0_sp*(sum(chint(2:n:2))-sum(chint(3:n:2)))   Set the constant of integration.
END FUNCTION chint
```

If you look at equation (5.9.1) for the Chebyshev coefficients of the integral of a function, you will see c_{i-1} and c_{i+1} and be tempted to use eoshift. We think it is almost always better to use array sections instead, as in the code above, especially if your code will ever run on a serial machine.

⋆ ⋆ ⋆

```
FUNCTION chebpc(c)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: c
REAL(SP), DIMENSION(size(c)) :: chebpc
```

Chebyshev polynomial coefficients. Given a coefficient array c of length N, this routine returns a coefficient array d of length N such that $\sum_{k=1}^{N} d_k y^{k-1} = \sum_{k=1}^{N} \mathtt{c}_k T_{k-1}(y) - \mathtt{c}_1/2$. The method is Clenshaw's recurrence (5.8.11), but now applied algebraically rather than arithmetically.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(c)) :: dd,sv
n=size(c)
chebpc=0.0
dd=0.0
chebpc(1)=c(n)
do j=n-1,2,-1
    sv(2:n-j+1)=chebpc(2:n-j+1)
    chebpc(2:n-j+1)=2.0_sp*chebpc(1:n-j)-dd(2:n-j+1)
    dd(2:n-j+1)=sv(2:n-j+1)
```

```
    sv(1)=chebpc(1)
    chebpc(1)=-dd(1)+c(j)
    dd(1)=sv(1)
end do
chebpc(2:n)=chebpc(1:n-1)-dd(2:n)
chebpc(1)=-dd(1)+0.5_sp*c(1)
END FUNCTION chebpc
```

⋆ ⋆ ⋆

```
SUBROUTINE pcshft(a,b,d)
USE nrtype; USE nrutil, ONLY : geop
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
```

Polynomial coefficient shift. Given a coefficient array d of length N, this routine generates a coefficient array g of the same length such that $\sum_{k=1}^{N} \mathrm{d}_k y^{k-1} = \sum_{k=1}^{N} g_k x^{k-1}$, where x and y are related by (5.8.10), i.e., the interval $-1 < y < 1$ is mapped to the interval $\mathrm{a} < x < \mathrm{b}$. The array g is returned in d.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(d)) :: dd
REAL(SP) :: x
n=size(d)
dd=d*geop(1.0_sp,2.0_sp/(b-a),n)
x=-0.5_sp*(a+b)
d(1)=dd(n)
d(2:n)=0.0
do j=n-1,1,-1
    d(2:n+1-j)=d(2:n+1-j)*x+d(1:n-j)
    d(1)=d(1)*x+dd(j)
end do
END SUBROUTINE pcshft
```

We accomplish the shift by synthetic division, that miracle of high-school algebra.

There is a subtle, but major, distinction between the synthetic division algorithm used in the Fortran 77 version of `pcshft` and that used above. In the Fortran 77 version, the synthetic division (translated to Fortran 90 notation) is

```
d(1:n)=dd(1:n)
do j=1,n-1
   do k=n-1,j,-1
      d(k)=x*d(k+1)+d(k)
   end do
end do
```

while, in Fortran 90, it is

```
d(1)=dd(n)
d(2:n)=0.0
do j=n-1,1,-1
   d(2:n+1-j)=d(2:n+1-j)*x+d(1:n-j)
   d(1)=d(1)*x+dd(j)
end do
```

As explained in §22.3, these are algebraically — but not algorithmically — equivalent. The inner loop in the Fortran 77 version does not parallelize, because each `k` value uses the result of the previous one. In fact, the `k` loop is a synthetic division, which can be parallelized *recursively* (as in the `nrutil` routine `poly_term`), but not simply

vectorized. In the Fortran 90 version, since not one but n−1 successive synthetic divisions are to be performed (by the outer loop), it is possible to reorganize the calculation to allow vectorization.

⋆ ⋆ ⋆

```
FUNCTION pccheb(d)
USE nrtype; USE nrutil, ONLY : arth,cumprod,geop
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP), DIMENSION(size(d)) :: pccheb
    Inverse of routine chebpc: given an array of polynomial coefficients d, returns an equivalent
    array of Chebyshev coefficients of the same length.
INTEGER(I4B) :: k,n
REAL(SP), DIMENSION(size(d)) :: denom,numer,pow
n=size(d)
pccheb(1)=2.0_sp*d(1)
pow=geop(1.0_sp,2.0_sp,n)              Powers of 2.
numer(1)=1.0                           Combinatorial coefficients computed as numer/denom.
denom(1)=1.0
denom(2:(n+3)/2)=cumprod(arth(1.0_sp,1.0_sp,(n+1)/2))
pccheb(2:n)=0.0
do k=2,n                               Loop over orders of x in the polynomial.
    numer(2:(k+3)/2)=cumprod(arth(k-1.0_sp,-1.0_sp,(k+1)/2))
    pccheb(k:1:-2)=pccheb(k:1:-2)+&
        d(k)/pow(k-1)*numer(1:(k+1)/2)/denom(1:(k+1)/2)
end do
END FUNCTION pccheb
```

⋆ ⋆ ⋆

```
SUBROUTINE pade(cof,resid)
USE nrtype
USE nr, ONLY : lubksb,ludcmp,mprove
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(INOUT) :: cof     DP for consistency with ratval.
REAL(SP), INTENT(OUT) :: resid
    Given cof(1:2N+1), the leading terms in the power series expansion of a function, solve
    the linear Padé equations to return the coefficients of a diagonal rational function approxi-
    mation to the same function, namely (cof(1) + cof(2)x + ··· + cof(N+1)x^N)/(1 +
    cof(N+2)x + ··· + cof(2N+1)x^N). The value resid is the norm of the residual
    vector; a small value indicates a well-converged solution.
INTEGER(I4B) :: k,n
INTEGER(I4B), DIMENSION((size(cof)-1)/2) :: indx
REAL(SP), PARAMETER :: BIG=1.0e30_sp     A big number.
REAL(SP) :: d,rr,rrold
REAL(SP), DIMENSION((size(cof)-1)/2) :: x,y,z
REAL(SP), DIMENSION((size(cof)-1)/2,(size(cof)-1)/2) :: q,qlu
n=(size(cof)-1)/2
x=cof(n+2:2*n+1)                         Set up matrix for solving.
y=x
do k=1,n
    q(:,k)=cof(n+2-k:2*n+1-k)
end do
qlu=q
call ludcmp(qlu,indx,d)                  Solve by LU decomposition and backsubsti-
call lubksb(qlu,indx,x)                      tution.
rr=BIG
do                                       Important to use iterative improvement, since
    rrold=rr                                 the Padé equations tend to be ill-conditioned.
```

```
    z=x
    call mprove(q,qlu,indx,y,x)
    rr=sum((z-x)**2)                    Calculate residual.
    if (rr >= rrold) exit               If it is no longer improving, call it quits.
end do
resid=sqrt(rrold)
do k=1,n                                Calculate the remaining coefficients.
    y(k)=cof(k+1)-dot_product(z(1:k),cof(k:1:-1))
end do
cof(2:n+1)=y                            Copy answers to output.
cof(n+2:2*n+1)=-z
END SUBROUTINE pade
```

⋆ ⋆ ⋆

```
SUBROUTINE ratlsq(func,a,b,mm,kk,cof,dev)
USE nrtype; USE nrutil, ONLY : arth,geop
USE nr, ONLY : ratval,svbksb,svdcmp
IMPLICIT NONE
REAL(DP), INTENT(IN) :: a,b
INTEGER(I4B), INTENT(IN) :: mm,kk
REAL(DP), DIMENSION(:), INTENT(OUT) :: cof
REAL(DP), INTENT(OUT) :: dev
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x
    REAL(DP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NPFAC=8,MAXIT=5
REAL(DP), PARAMETER :: BIG=1.0e30_dp
```

Returns in cof(1:mm+kk+1) the coefficients of a rational function approximation to the function func in the interval (a, b). Input quantities mm and kk specify the order of the numerator and denominator, respectively. The maximum absolute deviation of the approximation (insofar as is known) is returned as dev. Note that double-precision versions of svdcmp and svbksb are called.

```
INTEGER(I4B) :: it,ncof,npt,npth
REAL(DP) :: devmax,e,theta
REAL(DP), DIMENSION((mm+kk+1)*NPFAC) :: bb,ee,fs,wt,xs
REAL(DP), DIMENSION(mm+kk+1) :: coff,w
REAL(DP), DIMENSION(mm+kk+1,mm+kk+1) :: v
REAL(DP), DIMENSION((mm+kk+1)*NPFAC,mm+kk+1) :: u,temp
ncof=mm+kk+1
npt=NPFAC*ncof                          Number of points where function is evaluated,
npth=npt/2                                  i.e., fineness of the mesh.
dev=BIG
theta=PIO2_D/(npt-1)
xs(1:npth-1)=a+(b-a)*sin(theta*arth(0,1,npth-1))**2
```

Now fill arrays with mesh abscissas and function values. At each end, use formula that minimizes roundoff sensitivity in xs.

```
xs(npth:npt)=b-(b-a)*sin(theta*arth(npt-npth,-1,npt-npth+1))**2
fs=func(xs)
wt=1.0                                  In later iterations we will adjust these weights to
ee=1.0                                      combat the largest deviations.
e=0.0
do it=1,MAXIT                           Loop over iterations.
    bb=wt*(fs+sign(e,ee))
```

Key idea here: Fit to fn$(x) + e$ where the deviation is positive, to fn$(x) - e$ where it is negative. Then e is supposed to become an approximation to the equal-ripple deviation.

```
    temp=geop(spread(1.0_dp,1,npt),xs,ncof)
```

```
      Note that vector form of geop (returning matrix) is being used.
    u(:,1:mm+1)=temp(:,1:mm+1)*spread(wt,2,mm+1)
      Set up the "design matrix" for the least squares fit.
    u(:,mm+2:ncof)=-temp(:,2:ncof-mm)*spread(bb,2,ncof-mm-1)
    call svdcmp(u,w,v)
      Singular Value Decomposition. In especially singular or difficult cases, one might here
      edit the singular values w(1:ncof), replacing small values by zero.
    call svbksb(u,w,v,bb,coff)
    ee=ratval(xs,coff,mm,kk)-fs          Tabulate the deviations and revise the weights.
    wt=abs(ee)                           Use weighting to emphasize most deviant points.
    devmax=maxval(wt)
    e=sum(wt)/npt                        Update e to be the mean absolute deviation.
    if (devmax <= dev) then              Save only the best coefficient set found.
        cof=coff
        dev=devmax
    end if
    write(*,10) it,devmax
  end do
10 format (' ratlsq iteration=',i2,' max error=',1p,e10.3)
  END SUBROUTINE ratlsq
```

`temp=geop(spread(1.0_dp,1,npt),xs,ncof)` The design matrix u_{ij} is defined for $i = 1, \ldots,$ `npts` by

$$u_{ij} = \begin{cases} w_i x_i^{j-1}, & j = 1, \ldots, m+1 \\ -b_i x_i^{j-m-2}, & j = m+2, \ldots, n \end{cases} \tag{B5.12}$$

The first case in equation (B5.12) is computed in parallel by constructing the matrix `temp` equal to

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots \\ 1 & x_2 & x_2^2 & \cdots \\ 1 & x_3 & x_3^2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

and then multiplying by the matrix `spread(wt,2,mm+1)`, which is just

$$\begin{bmatrix} w_1 & w_1 & w_1 & \cdots \\ w_2 & w_2 & w_2 & \cdots \\ w_3 & w_3 & w_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

(Remember that multiplication using `*` means element-by-element multiplication, not matrix multiplication.) A similar construction is used for the second part of the design matrix.

Chapter B6. Special Functions

A Fortran 90 intrinsic function such as `sin(x)` is both *generic* and *elemental*. Generic means that the argument `x` can be any of multiple intrinsic data types and kind values (in the case of `sin`, any real or complex kind). Elemental means that `x` need not be a scalar, but can be an array of any rank and shape, in which case the calculation of `sin` is performed independently for each element.

Ideally, when we implement more complicated special functions in Fortran 90, as we do in this chapter, we would make them, too, both generic and elemental. Unfortunately, the language standard does not completely allow this. User-defined elemental functions are prohibited in Fortran 90, though they will be allowed in Fortran 95. And, there is no fully automatic way of providing for a single routine to allow arguments of multiple data types or kinds — nothing like C++'s "class templates," for example.

However, don't give up hope! Fortran 90 does provide a powerful mechanism for overloading, which can be used (perhaps not always with a maximum of convenience) to *simulate* both generic and elemental function features. In most cases, when we implement a special function with a scalar argument, `gammln(x)` say, we will also implement a corresponding vector-valued function of vector argument that evaluates the special function for each component of the vector argument. We will then overload the scalar and vector version of the function onto the same function name. For example, within the `nr` module are the lines

```
INTERFACE gammln
   FUNCTION gammln_s(xx)
   USE nrtype
   REAL(SP), INTENT(IN) :: xx
   REAL(SP) :: gammln_s
   END FUNCTION gammln_s

   FUNCTION gammln_v(xx)
   USE nrtype
   REAL(SP), DIMENSION(:), INTENT(IN) :: xx
   REAL(SP), DIMENSION(size(xx)) :: gammln_v
   END FUNCTION gammln_v
END INTERFACE
```

which can be included by a statement like "`USE nr, ONLY: gammln`," and then allow you to write `gammln(x)` without caring (or even thinking about) whether `x` is a scalar or a vector. If you want arguments of even higher rank (matrices, and so forth), you can provide these yourself, based on our models, and overload them, too.

That takes care of "elemental"; what about "generic"? Here, too, overloading provides an acceptable, if not perfect, solution. Where double-precision versions of special functions are needed, you can in many cases easily construct them from our provided routines by changing the variable kinds (and any necessary convergence

parameters), and then additionally overload them onto the same generic function names. (In general, in the interest of brevity, we will not ourselves do this for the functions in this chapter.)

At first meeting, Fortran 90's overloading capability may seem trivial, or merely cosmetic, to the Fortran 77 programmer; but one soon comes to rely on it as an important conceptual simplification. Programming at a "higher level of abstraction" is usually more productive than spending time "bogged down in the mud." Furthermore, the use of overloading is generally fail-safe: If you invoke a generic name with arguments of shapes or types for which a specific routine has not been defined, the compiler tells you about it.

We won't reprint the module `nr`'s interface blocks for all the routines in this chapter. When you see routines named `something_s` and `something_v`, below, you can safely assume that the generic name `something` is defined in the module `nr` and overloaded with the two specific routine names. A full alphabetical listing of all the interface blocks in `nr` is given in Appendix C2.

Given our heavy investment, in this chapter, in overloadable vector-valued special function routines, it is worth discussing whether this effort is simply a stopgap measure for Fortran 90, soon to be made obsolete by Fortran 95's provision of user-definable `ELEMENTAL` procedures. The answer is "not necessarily," and takes us into some speculation about the future of SIMD, versus MIMD, computing.

Elemental procedures, while applying the same executable code to each element, do not insist that it be feasible to perform all the parallel calculations in lockstep. That is, elemental procedures can have tests and branches (`if-then-else` constructions) that result in different elements being calculated by totally different pieces of code, in a fashion that can only be determined at run time. For true 100% MIMD (multiple instruction, multiple data) machines, this is not a problem: individual processors do the individual element calculations asynchronously.

However, virtually none of today's (and likely tomorrow's) largest-scale parallel supercomputers are 100% MIMD in this way. While modern parallel supercomputers increasingly have MIMD features, they continue to reward the use of SIMD (single instruction, multiple data) code with greater computational speed, often because of hardware pipelining or vector processing features within the individual processors. The use of Fortran 90 (or, for that matter Fortran 95) in a data-parallel or SIMD mode is thus by no means superfluous, or obviated by Fortran 95's `ELEMENTAL` construction.

The problem we face is that parallel calculation of special function values often doesn't fit well into the SIMD mold: Since the calculation of the value of a special function typically requires the convergence of an iterative process, as well as possible branches for different values of arguments, it cannot *in general* be done efficiently with "lockstep" SIMD programming.

Luckily, in particular cases, including most (but not all) of the functions in this chapter, one can in fact make reasonably good parallel implementations with the SIMD tools provided by the language. We will in fact see a number of different tricks for accomplishing this in the code that follows.

We are interested in demonstrating SIMD techniques, but we are not completely impractical. None of the data-parallel implementations given below are too inefficient on a scalar machine, and some may in fact be faster than Fortran 95's `ELEMENTAL`

alternative, or than do-loops over calls to the scalar version of the function. On a scalar machine, how can this be? We have already, above, hinted at the answer: (i) most modern scalar processors can overlap instructions to some degree, and data-parallel coding often provides compilers with the ability to accomplish this more efficiently; and (ii) data-parallel code can sometimes give better cache utilization.

⋆ ⋆ ⋆

```
FUNCTION gammln_s(xx)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xx
REAL(SP) :: gammln_s
    Returns the value ln[Γ(xx)] for xx > 0.
REAL(DP) :: tmp,x
    Internal arithmetic will be done in double precision, a nicety that you can omit if five-figure
    accuracy is good enough.
REAL(DP) :: stp = 2.5066282746310005_dp
REAL(DP), DIMENSION(6) :: coef = (/76.18009172947146_dp,&
    -86.50532032941677_dp,24.01409824083091_dp,&
    -1.231739572450155_dp,0.1208650973866179e-2_dp,&
    -0.5395239384953e-5_dp/)
call assert(xx > 0.0, 'gammln_s arg')
x=xx
tmp=x+5.5_dp
tmp=(x+0.5_dp)*log(tmp)-tmp
gammln_s=tmp+log(stp*(1.000000000190015_dp+&
    sum(coef(:)/arth(x+1.0_dp,1.0_dp,size(coef))))/x)
END FUNCTION gammln_s
```

```
FUNCTION gammln_v(xx)
USE nrtype; USE nrutil, ONLY: assert
IMPLICIT NONE
INTEGER(I4B) :: i
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
REAL(SP), DIMENSION(size(xx)) :: gammln_v
REAL(DP), DIMENSION(size(xx)) :: ser,tmp,x,y
REAL(DP) :: stp = 2.5066282746310005_dp
REAL(DP), DIMENSION(6) :: coef = (/76.18009172947146_dp,&
    -86.50532032941677_dp,24.01409824083091_dp,&
    -1.231739572450155_dp,0.1208650973866179e-2_dp,&
    -0.5395239384953e-5_dp/)
if (size(xx) == 0) RETURN
call assert(all(xx > 0.0), 'gammln_v arg')
x=xx
tmp=x+5.5_dp
tmp=(x+0.5_dp)*log(tmp)-tmp
ser=1.000000000190015_dp
y=x
do i=1,size(coef)
    y=y+1.0_dp
    ser=ser+coef(i)/y
end do
gammln_v=tmp+log(stp*ser/x)
END FUNCTION gammln_v
```

`call assert(xx > 0.0, 'gammln_s arg')` We use the `nrutil` routine `assert` for functions that have restrictions on the allowed range of arguments. One could instead have used an `if` statement with a call to `nrerror`; but we think that the uniformity of using `assert`, and the fact that its logical arguments read the "desired" way, not the "erroneous" way, make for a clearer programming style. In the vector version, the `assert` line is: `call assert(all(xx > 0.0), 'gammln_v arg')`

Notice that the scalar and vector versions achieve parallelism in quite different ways, something that we will see many times in this chapter. In the scalar case, parallelism (at least small-scale) is achieved through constructions like

```
sum(coef(:)/arth(x+1.0_dp,1.0_dp,size(coef)))
```

Here vector utilities construct the series $x+1, x+2, \ldots$ and then sum a series with these terms in the denominators and a vector of coefficients in the numerators. (This code may seem terse to Fortran 90 novices, but once you get used to it, it is quite clear to read.)

In the vector version, by contrast, parallelism is achieved across the components of the vector argument, and the above series is evaluated sequentially as a do-loop. Obviously the assumption is that the length of the vector argument is much longer than the very modest number (here, 6) of terms in the sum.

⋆ ⋆ ⋆

```
FUNCTION factrl_s(n)
USE nrtype; USE nrutil, ONLY : arth,assert,cumprod
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: factrl_s
    Returns the value n! as a floating-point number.
INTEGER(I4B), SAVE :: ntop=0
INTEGER(I4B), PARAMETER :: NMAX=32
REAL(SP), DIMENSION(NMAX), SAVE :: a        Table of stored values.
call assert(n >= 0, 'factrl_s arg')
if (n < ntop) then                          Already in table.
    factrl_s=a(n+1)
else if (n < NMAX) then                     Fill in table up to NMAX.
    ntop=NMAX
    a(1)=1.0
    a(2:NMAX)=cumprod(arth(1.0_sp,1.0_sp,NMAX-1))
    factrl_s=a(n+1)
else                                        Larger value than size of table is required.
    factrl_s=exp(gammln(n+1.0_sp))              Actually, this big a value is going to over-
end if                                          flow on many computers, but no harm in
END FUNCTION factrl_s                           trying.
```

`cumprod(arth(1.0_sp,1.0_sp,NMAX-1))` By now you should recognize this as an idiom for generating a vector of consecutive factorials. The routines `cumprod` and `arth`, both in `nrutil`, are both capable of being parallelized, e.g., by recursion, so this idiom is potentially faster than an in-line do-loop.

```
FUNCTION factrl_v(n)
USE nrtype; USE nrutil, ONLY : arth,assert,cumprod
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
REAL(SP), DIMENSION(size(n)) :: factrl_v
LOGICAL(LGT), DIMENSION(size(n)) :: mask
INTEGER(I4B), SAVE :: ntop=0
INTEGER(I4B), PARAMETER :: NMAX=32
REAL(SP), DIMENSION(NMAX), SAVE :: a
call assert(all(n >= 0), 'factrl_v arg')
if (ntop == 0) then
    ntop=NMAX
    a(1)=1.0
    a(2:NMAX)=cumprod(arth(1.0_sp,1.0_sp,NMAX-1))
end if
mask = (n >= NMAX)
factrl_v=unpack(exp(gammln(pack(n,mask)+1.0_sp)),mask,0.0_sp)
where (.not. mask) factrl_v=a(n+1)
END FUNCTION factrl_v
```

`unpack(exp(gammln(pack(n,mask)+1.0_sp)),mask,0.0_sp)` Here we meet the first of several solutions to a common problem: How shall we get answers, from an external vector-valued function, for just a *subset* of vector arguments, those defined by a mask? Here we use what we call the "pack-unpack" solution: Pack up all the arguments using the mask, send them to the function, and unpack the answers that come back. This packing and unpacking is not without cost (highly dependent on machine architecture, to be sure), but we hope to "earn it back" in the parallelism of the external function.

`where (.not. mask) factrl_v=a(n+1)` In some cases we might take care of the `.not.mask` case directly within the `unpack` construction, using its third ("`FIELD=`") argument to provide the not-unpacked values. However, there is no guarantee that the compiler won't evaluate all components of the "`FIELD=`" array, if it finds it efficient to do so. Here, since the index of `a(n+1)` would be out of range, we can't do it this way. Thus the separate `where` statement.

⋆ ⋆ ⋆

```
FUNCTION bico_s(n,k)
USE nrtype
USE nr, ONLY : factln
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,k
REAL(SP) :: bico_s
    Returns the binomial coefficient (n k) as a floating-point number.
bico_s=nint(exp(factln(n)-factln(k)-factln(n-k)))
  The nearest-integer function cleans up roundoff error for smaller values of n and k.
END FUNCTION bico_s
```

```
FUNCTION bico_v(n,k)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : factln
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n,k
REAL(SP), DIMENSION(size(n)) :: bico_v
INTEGER(I4B) :: ndum
ndum=assert_eq(size(n),size(k),'bico_v')
bico_v=nint(exp(factln(n)-factln(k)-factln(n-k)))
END FUNCTION bico_v
```

⋆ ⋆ ⋆

```
FUNCTION factln_s(n)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: factln_s
   Returns ln(n!).
INTEGER(I4B), PARAMETER :: TMAX=100
REAL(SP), DIMENSION(TMAX), SAVE :: a
LOGICAL(LGT), SAVE :: init=.true.
if (init) then                Initialize the table.
    a(1:TMAX)=gammln(arth(1.0_sp,1.0_sp,TMAX))
    init=.false.
end if
call assert(n >= 0, 'factln_s arg')
if (n < TMAX) then            In range of the table.
    factln_s=a(n+1)
else                          Out of range of the table.
    factln_s=gammln(n+1.0_sp)
end if
END FUNCTION factln_s
```

```
FUNCTION factln_v(n)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : gammln
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
REAL(SP), DIMENSION(size(n)) :: factln_v
LOGICAL(LGT), DIMENSION(size(n)) :: mask
INTEGER(I4B), PARAMETER :: TMAX=100
REAL(SP), DIMENSION(TMAX), SAVE :: a
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
    a(1:TMAX)=gammln(arth(1.0_sp,1.0_sp,TMAX))
    init=.false.
end if
call assert(all(n >= 0), 'factln_v arg')
mask = (n >= TMAX)
factln_v=unpack(gammln(pack(n,mask)+1.0_sp),mask,0.0_sp)
where (.not. mask) factln_v=a(n+1)
END FUNCTION factln_v
```

f90 gammln(arth(1.0_sp,1.0_sp,TMAX)) Another example of the programming convenience of combining a function returning a vector (here, arth) with a special function whose generic name (here, gammln) has an overloaded vector version.

⋆ ⋆ ⋆

```
FUNCTION beta_s(z,w)
USE nrtype
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: z,w
REAL(SP) :: beta_s
    Returns the value of the beta function B(z,w).
beta_s=exp(gammln(z)+gammln(w)-gammln(z+w))
END FUNCTION beta_s


FUNCTION beta_v(z,w)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: z,w
REAL(SP), DIMENSION(size(z)) :: beta_v
INTEGER(I4B) :: ndum
ndum=assert_eq(size(z),size(w),'beta_v')
beta_v=exp(gammln(z)+gammln(w)-gammln(z+w))
END FUNCTION beta_v
```

⋆ ⋆ ⋆

```
FUNCTION gammp_s(a,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP) :: gammp_s
    Returns the incomplete gamma function P(a,x).
call assert( x >= 0.0, a > 0.0, 'gammp_s args')
if (x<a+1.0_sp) then                Use the series representation.
    gammp_s=gser(a,x)
else                                Use the continued fraction representation
    gammp_s=1.0_sp-gcf(a,x)         and take its complement.
end if
END FUNCTION gammp_s


FUNCTION gammp_v(a,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
REAL(SP), DIMENSION(size(x)) :: gammp_v
LOGICAL(LGT), DIMENSION(size(x)) :: mask
INTEGER(I4B) :: ndum
ndum=assert_eq(size(a),size(x),'gammp_v')
call assert( all(x >= 0.0), all(a > 0.0), 'gammp_v args')
mask = (x<a+1.0_sp)
gammp_v=merge(gser(a,merge(x,0.0_sp,mask)), &
    1.0_sp-gcf(a,merge(x,0.0_sp,.not. mask)),mask)
END FUNCTION gammp_v
```

`call assert( x >= 0.0, a > 0.0, 'gammp_s args')` The generic routine `assert` in `nrutil` is overloaded with variants for more than one logical assertion, so you can make more than one assertion about argument ranges.

`gammp_v=merge(gser(a,merge(x,0.0_sp,mask)), &`
`1.0_sp-gcf(a,merge(x,0.0_sp,.not. mask)),mask)` Here we meet the *second* solution to the problem of getting masked values from an external vector function. (For the first solution, see note to `factrl`, above.) We call this one "merge with dummy values": Inappropriate values of the argument `x` (as determined by `mask`) are set to zero before `gser`, and later `gcf`, are called, and the supernumerary answers returned are discarded by a final `merge`. The assumption here is that the dummy value sent to the function (here, zero) is a special value that computes extremely fast, so that the overhead of computing and returning the supernumerary function values is outweighed by the parallelism achieved on the nontrivial components of `x`. Look at `gser_v` and `gcf_v` below to judge whether this assumption is realistic in this case.

```
FUNCTION gammq_s(a,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP) :: gammq_s
    Returns the incomplete gamma function Q(a,x) ≡ 1 − P(a,x).
call assert( x >= 0.0, a > 0.0, 'gammq_s args')
if (x<a+1.0_sp) then                Use the series representation
    gammq_s=1.0_sp-gser(a,x)        and take its complement.
else                                Use the continued fraction representation.
    gammq_s=gcf(a,x)
end if
END FUNCTION gammq_s



FUNCTION gammq_v(a,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : gcf,gser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
REAL(SP), DIMENSION(size(a)) :: gammq_v
LOGICAL(LGT), DIMENSION(size(x)) :: mask
INTEGER(I4B) :: ndum
ndum=assert_eq(size(a),size(x),'gammq_v')
call assert( all(x >= 0.0), all(a > 0.0), 'gammq_v args')
mask = (x<a+1.0_sp)
gammq_v=merge(1.0_sp-gser(a,merge(x,0.0_sp,mask)), &
    gcf(a,merge(x,0.0_sp,.not. mask)),mask)
END FUNCTION gammq_v



FUNCTION gser_s(a,x,gln)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP), OPTIONAL, INTENT(OUT) :: gln
REAL(SP) :: gser_s
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x)
```

Returns the incomplete gamma function $P(a,x)$ evaluated by its series representation as gamser. Also optionally returns $\ln\Gamma(a)$ as gln.

```
INTEGER(I4B) :: n
REAL(SP) :: ap,del,summ
if (x == 0.0) then
    gser_s=0.0
    RETURN
end if
ap=a
summ=1.0_sp/a
del=summ
do n=1,ITMAX
    ap=ap+1.0_sp
    del=del*x/ap
    summ=summ+del
    if (abs(del) < abs(summ)*EPS) exit
end do
if (n > ITMAX) call nrerror('a too large, ITMAX too small in gser_s')
if (present(gln)) then
    gln=gammln(a)
    gser_s=summ*exp(-x+a*log(x)-gln)
else
    gser_s=summ*exp(-x+a*log(x)-gammln(a))
end if
END FUNCTION gser_s



FUNCTION gser_v(a,x,gln)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
REAL(SP), DIMENSION(size(a)) :: gser_v
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x)
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(a)) :: ap,del,summ
LOGICAL(LGT), DIMENSION(size(a)) :: converged,zero
n=assert_eq(size(a),size(x),'gser_v')
zero=(x == 0.0)
where (zero) gser_v=0.0
ap=a
summ=1.0_sp/a
del=summ
converged=zero
do n=1,ITMAX
    where (.not. converged)
        ap=ap+1.0_sp
        del=del*x/ap
        summ=summ+del
        converged = (abs(del) < abs(summ)*EPS)
    end where
    if (all(converged)) exit
end do
if (n > ITMAX) call nrerror('a too large, ITMAX too small in gser_v')
if (present(gln)) then
    if (size(gln) < size(a)) call &
        nrerror('gser: Not enough space for gln')
    gln=gammln(a)
    where (.not. zero) gser_v=summ*exp(-x+a*log(x)-gln)
else
    where (.not. zero) gser_v=summ*exp(-x+a*log(x)-gammln(a))
```

```
end if
END FUNCTION gser_v
```

`REAL(SP), OPTIONAL, INTENT(OUT) :: gln` Normally, an `OPTIONAL` argument will be `INTENT(IN)` and be used to provide a less-often-used extra input argument to a function. Here, the `OPTIONAL` argument is `INTENT(OUT)`, used to provide a useful value that is a byproduct of the main calculation.

Also note that although $x \geq 0$ is required, we omit our usual `call assert` check for this, because `gser` is supposed to be called only by `gammp` or `gammq` — and these routines supply the argument checking themselves.

`do n=1,ITMAX...end do...if (n > ITMAX)...` This is typical code in Fortran 90 for a loop with a maximum number of iterations, relying on Fortran 90's guarantee that the index of the do-loop will be available after normal completion of the loop with a predictable value, greater by one than the upper limit of the loop. If the `exit` statement within the loop is ever taken, the `if` statement is guaranteed to fail; if the loop goes all the way through `ITMAX` cycles, the `if` statement is guaranteed to succeed.

`zero=(x == 0.0)...where (zero) gser_v=0.0...converged=zero` This is the code that provides for very low overhead calculation of zero arguments, as is assumed by the merge-with-dummy-values strategy in `gammp` and `gammq`. Zero arguments are "pre-converged" and are never the holdouts in the convergence test.

```
FUNCTION gcf_s(a,x,gln)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,x
REAL(SP), OPTIONAL, INTENT(OUT) :: gln
REAL(SP) :: gcf_s
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=tiny(x)/EPS
    Returns the incomplete gamma function Q(a,x) evaluated by its continued fraction representation as gammcf. Also optionally returns ln Γ(a) as gln.
    Parameters: ITMAX is the maximum allowed number of iterations; EPS is the relative accuracy; FPMIN is a number near the smallest representable floating-point number.
INTEGER(I4B) :: i
REAL(SP) :: an,b,c,d,del,h
if (x == 0.0) then
    gcf_s=1.0
    RETURN
end if
b=x+1.0_sp-a                          Set up for evaluating continued fraction by modified Lentz's method (§5.2) with b_0 = 0.
c=1.0_sp/FPMIN
d=1.0_sp/b
h=d
do i=1,ITMAX                          Iterate to convergence.
    an=-i*(i-a)
    b=b+2.0_sp
    d=an*d+b
    if (abs(d) < FPMIN) d=FPMIN
    c=b+an/c
    if (abs(c) < FPMIN) c=FPMIN
```

```
    d=1.0_sp/d
    del=d*c
    h=h*del
    if (abs(del-1.0_sp) <= EPS) exit
end do
if (i > ITMAX) call nrerror('a too large, ITMAX too small in gcf_s')
if (present(gln)) then
    gln=gammln(a)
    gcf_s=exp(-x+a*log(x)-gln)*h       Put factors in front.
else
    gcf_s=exp(-x+a*log(x)-gammln(a))*h
end if
END FUNCTION gcf_s


FUNCTION gcf_v(a,x,gln)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
REAL(SP), DIMENSION(size(a)) :: gcf_v
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=tiny(x)/EPS
INTEGER(I4B) :: i
REAL(SP), DIMENSION(size(a)) :: an,b,c,d,del,h
LOGICAL(LGT), DIMENSION(size(a)) :: converged,zero
i=assert_eq(size(a),size(x),'gcf_v')
zero=(x == 0.0)
where (zero)
    gcf_v=1.0
elsewhere
    b=x+1.0_sp-a
    c=1.0_sp/FPMIN
    d=1.0_sp/b
    h=d
end where
converged=zero
do i=1,ITMAX
    where (.not. converged)
        an=-i*(i-a)
        b=b+2.0_sp
        d=an*d+b
        d=merge(FPMIN,d, abs(d)<FPMIN )
        c=b+an/c
        c=merge(FPMIN,c, abs(c)<FPMIN )
        d=1.0_sp/d
        del=d*c
        h=h*del
        converged = (abs(del-1.0_sp)<=EPS)
    end where
    if (all(converged)) exit
end do
if (i > ITMAX) call nrerror('a too large, ITMAX too small in gcf_v')
if (present(gln)) then
    if (size(gln) < size(a)) call &
        nrerror('gser: Not enough space for gln')
    gln=gammln(a)
    where (.not. zero) gcf_v=exp(-x+a*log(x)-gln)*h
else
    where (.not. zero) gcf_v=exp(-x+a*log(x)-gammln(a))*h
end if
END FUNCTION gcf_v
```

`zero=(x == 0.0)...where (zero) gcf_v=1.0...converged=zero` See note on `gser`. Here, too, we pre-converge the special value of zero.

⋆ ⋆ ⋆

```
FUNCTION erf_s(x)
USE nrtype
USE nr, ONLY : gammp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erf_s
    Returns the error function erf(x).
erf_s=gammp(0.5_sp,x**2)
if (x < 0.0) erf_s=-erf_s
END FUNCTION erf_s
```

```
FUNCTION erf_v(x)
USE nrtype
USE nr, ONLY : gammp
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erf_v
erf_v=gammp(spread(0.5_sp,1,size(x)),x**2)
where (x < 0.0) erf_v=-erf_v
END FUNCTION erf_v
```

`erf_v=gammp(spread(0.5_sp,1,size(x)),x**2)` Yes, we do have an overloaded vector version of `gammp`, but it is vectorized on *both* its arguments. Thus, in a case where we want to vectorize on only *one* argument, we need a `spread` construction. In many contexts, Fortran 90 automatically makes scalars conformable with arrays (i.e., it automatically spreads them to the shape of the array); but the language does *not* do so when trying to match a generic function or subroutine call to a specific overloaded name. Perhaps this is wise; it is safer to prevent "accidental" invocations of vector-specific functions. Or, perhaps it is an area where the language could be improved.

```
FUNCTION erfc_s(x)
USE nrtype
USE nr, ONLY : gammp,gammq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erfc_s
    Returns the complementary error function erfc(x).
erfc_s=merge(1.0_sp+gammp(0.5_sp,x**2),gammq(0.5_sp,x**2), x < 0.0)
END FUNCTION erfc_s
```

`erfc_s=merge(1.0_sp+gammp(0.5_sp,x**2),gammq(0.5_sp,x**2), x < 0.0)` An example of our use of `merge` as an idiom for a conditional expression. Once you get used to these, you'll find them just as clear as the multiline `if...then...else` alternative.

```
FUNCTION erfc_v(x)
USE nrtype
USE nr, ONLY : gammp,gammq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erfc_v
LOGICAL(LGT), DIMENSION(size(x)) :: mask
mask = (x < 0.0)
erfc_v=merge(1.0_sp+gammp(spread(0.5_sp,1,size(x)), &
    merge(x,0.0_sp,mask)**2),gammq(spread(0.5_sp,1,size(x)), &
    merge(x,0.0_sp,.not. mask)**2),mask)
END FUNCTION erfc_v
```

`erfc_v=merge(1.0_sp+...)` Another example of the "merge with dummy values" idiom described on p. 1090. Here positive values of `x` in the call to `gammp`, and negative values in the call to `gammq`, are first set to the dummy value zero. The value zero is a special argument that computes very fast. The unwanted dummy function values are then discarded by the final outer `merge`.

⋆ ⋆ ⋆

```
FUNCTION erfcc_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: erfcc_s
    Returns the complementary error function erfc(x) with fractional error everywhere less than
    1.2 × 10^−7.
REAL(SP) :: t,z
REAL(SP), DIMENSION(10) :: coef = (/-1.26551223_sp,1.00002368_sp,&
    0.37409196_sp,0.09678418_sp,-0.18628806_sp,0.27886807_sp,&
    -1.13520398_sp,1.48851587_sp,-0.82215223_sp,0.17087277_sp/)
z=abs(x)
t=1.0_sp/(1.0_sp+0.5_sp*z)
erfcc_s=t*exp(-z*z+poly(t,coef))
if (x < 0.0) erfcc_s=2.0_sp-erfcc_s
END FUNCTION erfcc_s
```

```
FUNCTION erfcc_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: erfcc_v,t,z
REAL(SP), DIMENSION(10) :: coef = (/-1.26551223_sp,1.00002368_sp,&
    0.37409196_sp,0.09678418_sp,-0.18628806_sp,0.27886807_sp,&
    -1.13520398_sp,1.48851587_sp,-0.82215223_sp,0.17087277_sp/)
z=abs(x)
t=1.0_sp/(1.0_sp+0.5_sp*z)
erfcc_v=t*exp(-z*z+poly(t,coef))
where (x < 0.0) erfcc_v=2.0_sp-erfcc_v
END FUNCTION erfcc_v
```

f90 erfcc_v=t*exp(-z*z+poly(t,coef)) The vector code is identical to the scalar, because the nrutil routine poly has overloaded cases for the evaluation of a polynomial at a single value of the independent variable, and at multiple values. One *could* also overload a version with a matrix of coefficients whose columns could be used for the simultaneous evaluation of different polynomials at different values of independent variable. The point is that as long as there are differences in the shapes of at least one argument, the intended version of poly can be discerned by the compiler.

⋆ ⋆ ⋆

```
FUNCTION expint(n,x)
USE nrtype; USE nrutil, ONLY : arth,assert,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: expint
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x),BIG=huge(x)*EPS
    Evaluates the exponential integral E_n(x).
    Parameters: MAXIT is the maximum allowed number of iterations; EPS is the desired relative
    error, not smaller than the machine precision; BIG is a number near the largest representable
    floating-point number; EULER (in nrtype) is Euler's constant γ.
INTEGER(I4B) :: i,nm1
REAL(SP) :: a,b,c,d,del,fact,h
call assert(n >= 0, x >= 0.0, (x > 0.0 .or. n > 1), &
    'expint args')
if (n == 0) then                Special case.
    expint=exp(-x)/x
    RETURN
end if
nm1=n-1
if (x == 0.0) then              Another special case.
    expint=1.0_sp/nm1
else if (x > 1.0) then          Lentz's algorithm (§5.2).
    b=x+n
    c=BIG
    d=1.0_sp/b
    h=d
    do i=1,MAXIT
        a=-i*(nm1+i)
        b=b+2.0_sp
        d=1.0_sp/(a*d+b)        Denominators cannot be zero.
        c=b+a/c
        del=c*d
        h=h*del
        if (abs(del-1.0_sp) <= EPS) exit
    end do
    if (i > MAXIT) call nrerror('expint: continued fraction failed')
    expint=h*exp(-x)
else                            Evaluate series.
    if (nm1 /= 0) then          Set first term.
        expint=1.0_sp/nm1
    else
        expint=-log(x)-EULER
    end if
    fact=1.0
    do i=1,MAXIT
        fact=-fact*x/i
        if (i /= nm1) then
            del=-fact/(i-nm1)
```

```
        else                          ψ(n) appears here.
            del=fact*(-log(x)-EULER+sum(1.0_sp/arth(1,1,nm1)))
        end if
        expint=expint+del
        if (abs(del) < abs(expint)*EPS) exit
    end do
    if (i > MAXIT) call nrerror('expint: series failed')
end if
END FUNCTION expint
```

expint does not readily parallelize, and we thus don't provide a vector version. For syntactic convenience you could make a vector version with a do-loop over calls to this scalar version; or, in Fortran 95, you can of course make the function ELEMENTAL.

⋆ ⋆ ⋆

```
FUNCTION ei(x)
USE nrtype; USE nrutil, ONLY : assert,nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: ei
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=tiny(x)/EPS
```

Computes the exponential integral $\mathrm{Ei}(x)$ for $x > 0$.

Parameters: MAXIT is the maximum number of iterations allowed; EPS is the relative error, or absolute error near the zero of Ei at $x = 0.3725$; FPMIN is a number near the smallest representable floating-point number; EULER (in nrtype) is Euler's constant γ.

```
INTEGER(I4B) :: k
REAL(SP) :: fact,prev,sm,term
call assert(x > 0.0, 'ei arg')
if (x < FPMIN) then                   Special case: avoid failure of convergence test
    ei=log(x)+EULER                       because of underflow.
else if (x <= -log(EPS)) then         Use power series.
    sm=0.0
    fact=1.0
    do k=1,MAXIT
        fact=fact*x/k
        term=fact/k
        sm=sm+term
        if (term < EPS*sm) exit
    end do
    if (k > MAXIT) call nrerror('series failed in ei')
    ei=sm+log(x)+EULER
else                                  Use asymptotic series.
    sm=0.0                            Start with second term.
    term=1.0
    do k=1,MAXIT
        prev=term
        term=term*k/x
        if (term < EPS) exit          Since final sum is greater than one, term itself
        if (term < prev) then             approximates the relative error.
            sm=sm+term                Still converging: add new term.
        else                          Diverging: subtract previous term and exit.
            sm=sm-prev
            exit
        end if
    end do
    if (k > MAXIT) call nrerror('asymptotic failed in ei')
    ei=exp(x)*(1.0_sp+sm)/x
end if
END FUNCTION ei
```

ei does not readily parallelize, and we thus don't provide a vector version. For syntactic convenience you could make a vector version with a do-loop over calls to this scalar version; or, in Fortran 95, you can of course make the function ELEMENTAL.

⋆ ⋆ ⋆

```
FUNCTION betai_s(a,b,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : betacf,gammln
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,x
REAL(SP) :: betai_s
    Returns the incomplete beta function I_x(a,b).
REAL(SP) :: bt
call assert(x >= 0.0, x <= 1.0, 'betai_s arg')
if (x == 0.0 .or. x == 1.0) then
    bt=0.0
else                                          Factors in front of the continued fraction.
    bt=exp(gammln(a+b)-gammln(a)-gammln(b)&
        +a*log(x)+b*log(1.0_sp-x))
end if
if (x < (a+1.0_sp)/(a+b+2.0_sp)) then         Use continued fraction directly.
    betai_s=bt*betacf(a,b,x)/a
else                                          Use continued fraction after making the
    betai_s=1.0_sp-bt*betacf(b,a,1.0_sp-x)/b      symmetry transformation.
end if
END FUNCTION betai_s
```

```
FUNCTION betai_v(a,b,x)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : betacf,gammln
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
REAL(SP), DIMENSION(size(a)) :: betai_v
REAL(SP), DIMENSION(size(a)) :: bt
LOGICAL(LGT), DIMENSION(size(a)) :: mask
INTEGER(I4B) :: ndum
ndum=assert_eq(size(a),size(b),size(x),'betai_v')
call assert(all(x >= 0.0), all(x <= 1.0), 'betai_v arg')
where (x == 0.0 .or. x == 1.0)
    bt=0.0
elsewhere
    bt=exp(gammln(a+b)-gammln(a)-gammln(b)&
        +a*log(x)+b*log(1.0_sp-x))
end where
mask=(x < (a+1.0_sp)/(a+b+2.0_sp))
betai_v=bt*betacf(merge(a,b,mask),merge(b,a,mask),&
    merge(x,1.0_sp-x,mask))/merge(a,b,mask)
where (.not. mask) betai_v=1.0_sp-betai_v
END FUNCTION betai_v
```

Compare the scalar

```
if (x < (a+1.0_sp)/(a+b+2.0_sp)) then
   betai_s=bt*betacf(a,b,x)/a
else
   betai_s=1.0_sp-bt*betacf(b,a,1.0_sp-x)/b
end if
```

with the vector

```
mask=(x < (a+1.0_sp)/(a+b+2.0_sp))
betai_v=bt*betacf(merge(a,b,mask),merge(b,a,mask),&
   merge(x,1.0_sp-x,mask))/merge(a,b,mask)
where (.not. mask) betai_v=1.0_sp-betai_v
```

Here `merge` is used (several times) to evaluate all the required components in a single call to the vectorized `betacf`, notwithstanding that some components require one pattern of arguments, some a different pattern.

```
FUNCTION betacf_s(a,b,x)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,x
REAL(SP) :: betacf_s
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x), FPMIN=tiny(x)/EPS
   Used by betai: Evaluates continued fraction for incomplete beta function by modified
   Lentz's method (§5.2).
REAL(SP) :: aa,c,d,del,h,qab,qam,qap
INTEGER(I4B) :: m,m2
qab=a+b                                  These q's will be used in factors that occur
qap=a+1.0_sp                                in the coefficients (6.4.6).
qam=a-1.0_sp
c=1.0                                    First step of Lentz's method.
d=1.0_sp-qab*x/qap
if (abs(d) < FPMIN) d=FPMIN
d=1.0_sp/d
h=d
do m=1,MAXIT
    m2=2*m
    aa=m*(b-m)*x/((qam+m2)*(a+m2))
    d=1.0_sp+aa*d                        One step (the even one) of the recurrence.
    if (abs(d) < FPMIN) d=FPMIN
    c=1.0_sp+aa/c
    if (abs(c) < FPMIN) c=FPMIN
    d=1.0_sp/d
    h=h*d*c
    aa=-(a+m)*(qab+m)*x/((a+m2)*(qap+m2))
    d=1.0_sp+aa*d                        Next step of the recurrence (the odd one).
    if (abs(d) < FPMIN) d=FPMIN
    c=1.0_sp+aa/c
    if (abs(c) < FPMIN) c=FPMIN
    d=1.0_sp/d
    del=d*c
    h=h*del
    if (abs(del-1.0_sp) <= EPS) exit     Are we done?
end do
if (m > MAXIT)&
    call nrerror('a or b too big, or MAXIT too small in betacf_s')
betacf_s=h
END FUNCTION betacf_s
```

```
FUNCTION betacf_v(a,b,x)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
REAL(SP), DIMENSION(size(x)) :: betacf_v
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x), FPMIN=tiny(x)/EPS
REAL(SP), DIMENSION(size(x)) :: aa,c,d,del,h,qab,qam,qap
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(I4B) :: m
INTEGER(I4B), DIMENSION(size(x)) :: m2
m=assert_eq(size(a),size(b),size(x),'betacf_v')
qab=a+b
qap=a+1.0_sp
qam=a-1.0_sp
c=1.0
d=1.0_sp-qab*x/qap
where (abs(d) < FPMIN) d=FPMIN
d=1.0_sp/d
h=d
converged=.false.
do m=1,MAXIT
    where (.not. converged)
        m2=2*m
        aa=m*(b-m)*x/((qam+m2)*(a+m2))
        d=1.0_sp+aa*d
        d=merge(FPMIN,d, abs(d)<FPMIN )
        c=1.0_sp+aa/c
        c=merge(FPMIN,c, abs(c)<FPMIN )
        d=1.0_sp/d
        h=h*d*c
        aa=-(a+m)*(qab+m)*x/((a+m2)*(qap+m2))
        d=1.0_sp+aa*d
        d=merge(FPMIN,d, abs(d)<FPMIN )
        c=1.0_sp+aa/c
        c=merge(FPMIN,c, abs(c)<FPMIN )
        d=1.0_sp/d
        del=d*c
        h=h*del
        converged = (abs(del-1.0_sp) <= EPS)
    end where
    if (all(converged)) exit
end do
if (m > MAXIT)&
    call nrerror('a or b too big, or MAXIT too small in betacf_v')
betacf_v=h
END FUNCTION betacf_v
```

f90 `d=merge(FPMIN,d, abs(d)<FPMIN )` The scalar version does this with an `if`. Why does it become a `merge` here in the vector version, rather than a `where`? Because we are already inside a "`where (.not.converged)`" block, and Fortran 90 doesn't allow nested `where`'s! (Fortran 95 *will* allow nested `where`'s.)

⋆ ⋆ ⋆

```
FUNCTION bessj0_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj0_s
    Returns the Bessel function J_0(x) for any real x.
REAL(SP) :: ax,xx,z
REAL(DP) :: y                          We'll accumulate polynomials in double precision.
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
    0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/-0.1562499995e-1_dp,&
    0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
    -0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/57568490574.0_dp,-13362590354.0_dp,&
    651619640.7_dp,-11214424.18_dp,77392.33017_dp,&
    -184.9052456_dp/)
REAL(DP), DIMENSION(6) :: s = (/57568490411.0_dp,1029532985.0_dp,&
    9494680.718_dp,59272.64853_dp,267.8532712_dp,1.0_dp/)
if (abs(x) < 8.0) then                 Direct rational function fit.
    y=x**2
    bessj0_s=poly(y,r)/poly(y,s)
else                                   Fitting function (6.5.9).
    ax=abs(x)
    z=8.0_sp/ax
    y=z**2
    xx=ax-0.785398164_sp
    bessj0_s=sqrt(0.636619772_sp/ax)*(cos(xx)*&
        poly(y,p)-z*sin(xx)*poly(y,q))
end if
END FUNCTION bessj0_s


FUNCTION bessj0_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj0_v
REAL(SP), DIMENSION(size(x)) :: ax,xx,z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
    0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/-0.1562499995e-1_dp,&
    0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
    -0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/57568490574.0_dp,-13362590354.0_dp,&
    651619640.7_dp,-11214424.18_dp,77392.33017_dp,&
    -184.9052456_dp/)
REAL(DP), DIMENSION(6) :: s = (/57568490411.0_dp,1029532985.0_dp,&
    9494680.718_dp,59272.64853_dp,267.8532712_dp,1.0_dp/)
mask = (abs(x) < 8.0)
where (mask)
    y=x**2
    bessj0_v=poly(y,r,mask)/poly(y,s,mask)
elsewhere
    ax=abs(x)
    z=8.0_sp/ax
    y=z**2
    xx=ax-0.785398164_sp
    bessj0_v=sqrt(0.636619772_sp/ax)*(cos(xx)*&
        poly(y,p,.not. mask)-z*sin(xx)*poly(y,q,.not. mask))
end where
END FUNCTION bessj0_v
```

`where (mask)...bessj0_v=poly(y,r,mask)/poly(y,s,mask)` Here we meet the *third* solution to the problem of getting masked values from an external vector function. (For the other two solutions, see notes to `factrl`, p. 1087, and `gammp`, p. 1090.) Here we simply evade all responsibility and pass the mask into every routine that is supposed to be masked. Let it be somebody else's problem! That works here because your hardworking authors have overloaded the `nrutil` routine `poly` with a masked vector version. More typically, of course, it becomes *your* problem, and you have to remember to write masked versions of all the vector routines that you call in this way. (We'll meet examples of this later.)

⋆ ⋆ ⋆

```
FUNCTION bessy0_s(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessj0
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessy0_s
   Returns the Bessel function Y_0(x) for positive x.
REAL(SP) :: xx,z
REAL(DP) :: y                    We'll accumulate polynomials in double precision.
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
    0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/-0.1562499995e-1_dp,&
    0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
    -0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/-2957821389.0_dp,7062834065.0_dp,&
    -512359803.6_dp,10879881.29_dp,-86327.92757_dp,&
    228.4622733_dp/)
REAL(DP), DIMENSION(6) :: s = (/40076544269.0_dp,745249964.8_dp,&
    7189466.438_dp,47447.26470_dp,226.1030244_dp,1.0_dp/)
call assert(x > 0.0, 'bessy0_s arg')
if (abs(x) < 8.0) then           Rational function approximation of (6.5.8).
    y=x**2
    bessy0_s=(poly(y,r)/poly(y,s))+&
        0.636619772_sp*bessj0(x)*log(x)
else                             Fitting function (6.5.10).
    z=8.0_sp/x
    y=z**2
    xx=x-0.785398164_sp
    bessy0_s=sqrt(0.636619772_sp/x)*(sin(xx)*&
        poly(y,p)+z*cos(xx)*poly(y,q))
end if
END FUNCTION bessy0_s
```

```
FUNCTION bessy0_v(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessj0
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessy0_v
REAL(SP), DIMENSION(size(x)) :: xx,z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,-0.1098628627e-2_dp,&
    0.2734510407e-4_dp,-0.2073370639e-5_dp,0.2093887211e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/-0.1562499995e-1_dp,&
    0.1430488765e-3_dp,-0.6911147651e-5_dp,0.7621095161e-6_dp,&
```

```
    -0.934945152e-7_dp/)
REAL(DP), DIMENSION(6) :: r = (/-2957821389.0_dp,7062834065.0_dp,&
    -512359803.6_dp,10879881.29_dp,-86327.92757_dp,&
    228.4622733_dp/)
REAL(DP), DIMENSION(6) :: s = (/40076544269.0_dp,745249964.8_dp,&
    7189466.438_dp,47447.26470_dp,226.1030244_dp,1.0_dp/)
call assert(all(x > 0.0), 'bessy0_v arg')
mask = (abs(x) < 8.0)
where (mask)
    y=x**2
    bessy0_v=(poly(y,r,mask)/poly(y,s,mask))+&
        0.636619772_sp*bessj0(x)*log(x)
elsewhere
    z=8.0_sp/x
    y=z**2
    xx=x-0.785398164_sp
    bessy0_v=sqrt(0.636619772_sp/x)*(sin(xx)*&
        poly(y,p,.not. mask)+z*cos(xx)*poly(y,q,.not. mask))
end where
END FUNCTION bessy0_v
```

⋆ ⋆ ⋆

```
FUNCTION bessj1_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj1_s
   Returns the Bessel function J_1(x) for any real x.
REAL(SP) :: ax,xx,z
REAL(DP) :: y                    We'll accumulate polynomials in double precision.
REAL(DP), DIMENSION(6) :: r = (/72362614232.0_dp,&
    -7895059235.0_dp,242396853.1_dp,-2972611.439_dp,&
    15704.48260_dp,-30.16036606_dp/)
REAL(DP), DIMENSION(6) :: s = (/144725228442.0_dp,2300535178.0_dp,&
    18583304.74_dp,99447.43394_dp,376.9991397_dp,1.0_dp/)
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,0.183105e-2_dp,&
    -0.3516396496e-4_dp,0.2457520174e-5_dp,-0.240337019e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp,&
    -0.2002690873e-3_dp,0.8449199096e-5_dp,-0.88228987e-6_dp,&
    0.105787412e-6_dp/)
if (abs(x) < 8.0) then           Direct rational approximation.
    y=x**2
    bessj1_s=x*(poly(y,r)/poly(y,s))
else                             Fitting function (6.5.9).
    ax=abs(x)
    z=8.0_sp/ax
    y=z**2
    xx=ax-2.356194491_sp
    bessj1_s=sqrt(0.636619772_sp/ax)*(cos(xx)*&
        poly(y,p)-z*sin(xx)*poly(y,q))*sign(1.0_sp,x)
end if
END FUNCTION bessj1_s
```

```
FUNCTION bessj1_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessj1_v
REAL(SP), DIMENSION(size(x)) :: ax,xx,z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(6) :: r = (/72362614232.0_dp,&
    -7895059235.0_dp,242396853.1_dp,-2972611.439_dp,&
    15704.48260_dp,-30.16036606_dp/)
REAL(DP), DIMENSION(6) :: s = (/144725228442.0_dp,2300535178.0_dp,&
    18583304.74_dp,99447.43394_dp,376.9991397_dp,1.0_dp/)
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,0.183105e-2_dp,&
    -0.3516396496e-4_dp,0.2457520174e-5_dp,-0.240337019e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp,&
    -0.2002690873e-3_dp,0.8449199096e-5_dp,-0.88228987e-6_dp,&
    0.105787412e-6_dp/)
mask = (abs(x) < 8.0)
where (mask)
    y=x**2
    bessj1_v=x*(poly(y,r,mask)/poly(y,s,mask))
elsewhere
    ax=abs(x)
    z=8.0_sp/ax
    y=z**2
    xx=ax-2.356194491_sp
    bessj1_v=sqrt(0.636619772_sp/ax)*(cos(xx)*&
        poly(y,p,.not. mask)-z*sin(xx)*poly(y,q,.not. mask))*&
        sign(1.0_sp,x)
end where
END FUNCTION bessj1_v
```

⋆ ⋆ ⋆

```
FUNCTION bessy1_s(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessj1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessy1_s
   Returns the Bessel function Y_1(x) for positive x.
REAL(SP) :: xx,z
REAL(DP) :: y                    We'll accumulate polynomials in double precision.
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,0.183105e-2_dp,&
    -0.3516396496e-4_dp,0.2457520174e-5_dp,-0.240337019e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp,&
    -0.2002690873e-3_dp,0.8449199096e-5_dp,-0.88228987e-6_dp,&
    0.105787412e-6_dp/)
REAL(DP), DIMENSION(6) :: r = (/-0.4900604943e13_dp,&
    0.1275274390e13_dp,-0.5153438139e11_dp,0.7349264551e9_dp,&
    -0.4237922726e7_dp,0.8511937935e4_dp/)
REAL(DP), DIMENSION(7) :: s = (/0.2499580570e14_dp,&
    0.4244419664e12_dp,0.3733650367e10_dp,0.2245904002e8_dp,&
    0.1020426050e6_dp,0.3549632885e3_dp,1.0_dp/)
call assert(x > 0.0, 'bessy1_s arg')
if (abs(x) < 8.0) then           Rational function approximation of (6.5.8).
    y=x**2
    bessy1_s=x*(poly(y,r)/poly(y,s))+&
        0.636619772_sp*(bessj1(x)*log(x)-1.0_sp/x)
else                             Fitting function (6.5.10).
```

```
    z=8.0_sp/x
    y=z**2
    xx=x-2.356194491_sp
    bessy1_s=sqrt(0.636619772_sp/x)*(sin(xx)*&
        poly(y,p)+z*cos(xx)*poly(y,q))
end if
END FUNCTION bessy1_s



FUNCTION bessy1_v(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessj1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessy1_v
REAL(SP), DIMENSION(size(x)) :: xx,z
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(5) :: p = (/1.0_dp,0.183105e-2_dp,&
    -0.3516396496e-4_dp,0.2457520174e-5_dp,-0.240337019e-6_dp/)
REAL(DP), DIMENSION(5) :: q = (/0.04687499995_dp,&
    -0.2002690873e-3_dp,0.8449199096e-5_dp,-0.88228987e-6_dp,&
    0.105787412e-6_dp/)
REAL(DP), DIMENSION(6) :: r = (/-0.4900604943e13_dp,&
    0.1275274390e13_dp,-0.5153438139e11_dp,0.7349264551e9_dp,&
    -0.4237922726e7_dp,0.8511937935e4_dp/)
REAL(DP), DIMENSION(7) :: s = (/0.2499580570e14_dp,&
    0.4244419664e12_dp,0.3733650367e10_dp,0.2245904002e8_dp,&
    0.1020426050e6_dp,0.3549632885e3_dp,1.0_dp/)
call assert(all(x > 0.0), 'bessy1_v arg')
mask = (abs(x) < 8.0)
where (mask)
    y=x**2
    bessy1_v=x*(poly(y,r,mask)/poly(y,s,mask))+&
        0.636619772_sp*(bessj1(x)*log(x)-1.0_sp/x)
elsewhere
    z=8.0_sp/x
    y=z**2
    xx=x-2.356194491_sp
    bessy1_v=sqrt(0.636619772_sp/x)*(sin(xx)*&
        poly(y,p,.not. mask)+z*cos(xx)*poly(y,q,.not. mask))
end where
END FUNCTION bessy1_v
```

⋆ ⋆ ⋆

```
FUNCTION bessy_s(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessy0,bessy1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessy_s
```

Returns the Bessel function $Y_{\mathtt{n}}(\mathtt{x})$ for positive x and $\mathtt{n} \geq 2$.

```
INTEGER(I4B) :: j
REAL(SP) :: by,bym,byp,tox
call assert(n >= 2, x > 0.0, 'bessy_s args')
tox=2.0_sp/x
by=bessy1(x)                    Starting values for the recurrence.
bym=bessy0(x)
do j=1,n-1                      Recurrence (6.5.7).
```

```
    byp=j*tox*by-bym
    bym=by
    by=byp
end do
bessy_s=by
END FUNCTION bessy_s



FUNCTION bessy_v(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessy0,bessy1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessy_v
INTEGER(I4B) :: j
REAL(SP), DIMENSION(size(x)) :: by,bym,byp,tox
call assert(n >= 2, all(x > 0.0), 'bessy_v args')
tox=2.0_sp/x
by=bessy1(x)
bym=bessy0(x)
do j=1,n-1
    byp=j*tox*by-bym
    bym=by
    by=byp
end do
bessy_v=by
END FUNCTION bessy_v
```

f90 Notice that the vector routine is *exactly* the same as the scalar routine, but operates only on vectors, and that nothing in the routine is specific to any level of precision or kind type of real variable. Cases like this make us wish that Fortran 90 provided for "template" types that could automatically take the type and shape of the actual arguments. (Such facilities are available in other, more object-oriented languages such as C++.)

⋆ ⋆ ⋆

```
FUNCTION bessj_s(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessj0,bessj1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessj_s
INTEGER(I4B), PARAMETER :: IACC=40,IEXP=maxexponent(x)/2
    Returns the Bessel function J_n(x) for any real x and n ≥ 2. Make the parameter IACC
    larger to increase accuracy.
INTEGER(I4B) :: j,jsum,m
REAL(SP) :: ax,bj,bjm,bjp,summ,tox
call assert(n >= 2, 'bessj_s args')
ax=abs(x)
if (ax*ax <= 8.0_sp*tiny(x)) then            Underflow limit.
    bessj_s=0.0
else if (ax > real(n,sp)) then               Upwards recurrence from J_0 and J_1.
    tox=2.0_sp/ax
    bjm=bessj0(ax)
    bj=bessj1(ax)
    do j=1,n-1
```

```
        bjp=j*tox*bj-bjm
        bjm=bj
        bj=bjp
    end do
    bessj_s=bj
else                                        Downwards recurrence from an even m
    tox=2.0_sp/ax                               here computed.
    m=2*((n+int(sqrt(real(IACC*n,sp))))/2)
    bessj_s=0.0
    jsum=0                                  jsum will alternate between 0 and 1; when
    summ=0.0                                    it is 1, we accumulate in sum the
    bjp=0.0                                     even terms in (5.5.16).
    bj=1.0
    do j=m,1,-1                             The downward recurrence.
        bjm=j*tox*bj-bjp
        bjp=bj
        bj=bjm
        if (exponent(bj) > IEXP) then       Renormalize to prevent overflows.
            bj=scale(bj,-IEXP)
            bjp=scale(bjp,-IEXP)
            bessj_s=scale(bessj_s,-IEXP)
            summ=scale(summ,-IEXP)
        end if
        if (jsum /= 0) summ=summ+bj         Accumulate the sum.
        jsum=1-jsum                         Change 0 to 1 or vice versa.
        if (j == n) bessj_s=bjp             Save the unnormalized answer.
    end do
    summ=2.0_sp*summ-bj                     Compute (5.5.16)
    bessj_s=bessj_s/summ                    and use it to normalize the answer.
end if
if (x < 0.0 .and. mod(n,2) == 1) bessj_s=-bessj_s
END FUNCTION bessj_s
```

The `bessj` routine does not conveniently parallelize with Fortran 90's language constructions, but Bessel functions are of sufficient importance that we feel the need for a parallel version nevertheless. The basic method adopted below is to encapsulate as contained vector functions two separate algorithms, one for the case $x \leq n$, the other for $x > n$. Both of these have masks as input arguments; within each routine, however, they immediately revert to the pack-unpack method. The choice to pack in the subsidiary routines, rather than in the main routine, is arbitrary; the main routine is supposed to be a little clearer this way.

`if (exponent(bj) > IEXP) then...` In the Fortran 77 version of this routine, we scaled the variables by 10^{-10} whenever `bj` was bigger than 10^{10}. On a machine with a large exponent range, we could improve efficiency by scaling less often. In order to remain portable, however, we used the conservative value of 10^{10}. An elegant way of handling renormalization is provided by the Fortran 90 intrinsic functions that manipulate real numbers. We test with `if (exponent(bj) > IEXP)` and then if necessary renormalize with `bj=scale(bj,-IEXP)` and similarly for the other variables. Our conservative choice is to set `IEXP=maxexponent(x)/2`. Note that an added benefit of scaling this way is that only the exponent of each variable is modified; no roundoff error is introduced as it can be if we do a floating-point division instead.

```
FUNCTION bessj_v(n,xx)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessj0,bessj1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
REAL(SP), DIMENSION(size(xx)) :: bessj_v
INTEGER(I4B), PARAMETER :: IACC=40,IEXP=maxexponent(xx)/2
REAL(SP), DIMENSION(size(xx)) :: ax
LOGICAL(LGT), DIMENSION(size(xx)) :: mask,mask0
REAL(SP), DIMENSION(:), ALLOCATABLE :: x,bj,bjm,bjp,summ,tox,bessjle
LOGICAL(LGT), DIMENSION(:), ALLOCATABLE :: renorm
INTEGER(I4B) :: j,jsum,m,npak
call assert(n >= 2, 'bessj_v args')
ax=abs(xx)
mask = (ax <= real(n,sp))
mask0 = (ax*ax <= 8.0_sp*tiny(xx))
bessj_v=bessjle_v(n,ax,logical(mask .and. .not.mask0, kind=lgt))
bessj_v=merge(bessjgt_v(n,ax,.not. mask),bessj_v,.not. mask)
where (mask0) bessj_v=0.0
where (xx < 0.0 .and. mod(n,2) == 1) bessj_v=-bessj_v
CONTAINS

FUNCTION bessjgt_v(n,xx,mask)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
LOGICAL(LGT), DIMENSION(size(xx)), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(xx)) :: bessjgt_v
npak=count(mask)
if (npak == 0) RETURN
allocate(x(npak),bj(npak),bjm(npak),bjp(npak),tox(npak))
x=pack(xx,mask)
tox=2.0_sp/x
bjm=bessj0(x)
bj=bessj1(x)
do j=1,n-1
    bjp=j*tox*bj-bjm
    bjm=bj
    bj=bjp
end do
bessjgt_v=unpack(bj,mask,0.0_sp)
deallocate(x,bj,bjm,bjp,tox)
END FUNCTION bessjgt_v

FUNCTION bessjle_v(n,xx,mask)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: xx
LOGICAL(LGT), DIMENSION(size(xx)), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(xx)) :: bessjle_v
npak=count(mask)
if (npak == 0) RETURN
allocate(x(npak),bj(npak),bjm(npak),bjp(npak),summ(npak), &
    bessjle(npak),tox(npak),renorm(npak))
x=pack(xx,mask)
tox=2.0_sp/x
m=2*((n+int(sqrt(real(IACC*n,sp))))/2)
bessjle=0.0
jsum=0
summ=0.0
bjp=0.0
bj=1.0
do j=m,1,-1
    bjm=j*tox*bj-bjp
```

```
    bjp=bj
    bj=bjm
    renorm = (exponent(bj)>IEXP)
    bj=merge(scale(bj,-IEXP),bj,renorm)
    bjp=merge(scale(bjp,-IEXP),bjp,renorm)
    bessjle=merge(scale(bessjle,-IEXP),bessjle,renorm)
    summ=merge(scale(summ,-IEXP),summ,renorm)
    if (jsum /= 0) summ=summ+bj
    jsum=1-jsum
    if (j == n) bessjle=bjp
end do
summ=2.0_sp*summ-bj
bessjle=bessjle/summ
bessjle_v=unpack(bessjle,mask,0.0_sp)
deallocate(x,bj,bjm,bjp,summ,bessjle,tox,renorm)
END FUNCTION bessjle_v
END FUNCTION bessj_v
```

f90 `bessj_v=... bessj_v=merge(bessjgt_v(...),bessj_v,...)` The vector `bessj_v` is set once (with a mask) and then merged with *itself*, along with the vector result of the `bessjgt_v` call. Thus are the two evaluation methods combined. (A third case, where an argument is zero, is then handled by an immediately following `where`.)

⋆ ⋆ ⋆

```
FUNCTION bessi0_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi0_s
    Returns the modified Bessel function I_0(x) for any real x.
REAL(SP) :: ax
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,3.5156229_dp,&
    3.0899424_dp,1.2067492_dp,0.2659732_dp,0.360768e-1_dp,&
    0.45813e-2_dp/)          Accumulate polynomials in double precision.
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp,0.1328592e-1_dp,&
    0.225319e-2_dp,-0.157565e-2_dp,0.916281e-2_dp,&
    -0.2057706e-1_dp,0.2635537e-1_dp,-0.1647633e-1_dp,&
    0.392377e-2_dp/)
ax=abs(x)
if (ax < 3.75) then          Polynomial fit.
    bessi0_s=poly(real((x/3.75_sp)**2,dp),p)
else
    bessi0_s=(exp(ax)/sqrt(ax))*poly(real(3.75_sp/ax,dp),q)
end if
END FUNCTION bessi0_s



FUNCTION bessi0_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi0_v
REAL(SP), DIMENSION(size(x)) :: ax
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,3.5156229_dp,&
    3.0899424_dp,1.2067492_dp,0.2659732_dp,0.360768e-1_dp,&
```

```
    0.45813e-2_dp/)
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp,0.1328592e-1_dp,&
    0.225319e-2_dp,-0.157565e-2_dp,0.916281e-2_dp,&
    -0.2057706e-1_dp,0.2635537e-1_dp,-0.1647633e-1_dp,&
    0.392377e-2_dp/)
ax=abs(x)
mask = (ax < 3.75)
where (mask)
    bessi0_v=poly(real((x/3.75_sp)**2,dp),p,mask)
elsewhere
    y=3.75_sp/ax
    bessi0_v=(exp(ax)/sqrt(ax))*poly(real(y,dp),q,.not. mask)
end where
END FUNCTION bessi0_v
```

⋆ ⋆ ⋆

```
FUNCTION bessk0_s(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessi0
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessk0_s
    Returns the modified Bessel function K_0(x) for positive real x.
REAL(DP) :: y                Accumulate polynomials in double precision.
REAL(DP), DIMENSION(7) :: p = (/-0.57721566_dp,0.42278420_dp,&
    0.23069756_dp,0.3488590e-1_dp,0.262698e-2_dp,0.10750e-3_dp,&
    0.74e-5_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,-0.7832358e-1_dp,&
    0.2189568e-1_dp,-0.1062446e-1_dp,0.587872e-2_dp,&
    -0.251540e-2_dp,0.53208e-3_dp/)
call assert(x > 0.0, 'bessk0_s arg')
if (x <= 2.0) then           Polynomial fit.
    y=x*x/4.0_sp
    bessk0_s=(-log(x/2.0_sp)*bessi0(x))+poly(y,p)
else
    y=(2.0_sp/x)
    bessk0_s=(exp(-x)/sqrt(x))*poly(y,q)
end if
END FUNCTION bessk0_s
```

```
FUNCTION bessk0_v(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessi0
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessk0_v
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(7) :: p = (/-0.57721566_dp,0.42278420_dp,&
    0.23069756_dp,0.3488590e-1_dp,0.262698e-2_dp,0.10750e-3_dp,&
    0.74e-5_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,-0.7832358e-1_dp,&
    0.2189568e-1_dp,-0.1062446e-1_dp,0.587872e-2_dp,&
    -0.251540e-2_dp,0.53208e-3_dp/)
call assert(all(x > 0.0), 'bessk0_v arg')
mask = (x <= 2.0)
where (mask)
    y=x*x/4.0_sp
    bessk0_v=(-log(x/2.0_sp)*bessi0(x))+poly(y,p,mask)
```

```
elsewhere
    y=(2.0_sp/x)
    bessk0_v=(exp(-x)/sqrt(x))*poly(y,q,.not. mask)
end where
END FUNCTION bessk0_v
```

⋆ ⋆ ⋆

```
FUNCTION bessi1_s(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi1_s
    Returns the modified Bessel function I_1(x) for any real x.
REAL(SP) :: ax
REAL(DP), DIMENSION(7) :: p = (/0.5_dp,0.87890594_dp,&
    0.51498869_dp,0.15084934_dp,0.2658733e-1_dp,&
    0.301532e-2_dp,0.32411e-3_dp/)
      Accumulate polynomials in double precision.
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp,-0.3988024e-1_dp,&
    -0.362018e-2_dp,0.163801e-2_dp,-0.1031555e-1_dp,&
    0.2282967e-1_dp,-0.2895312e-1_dp,0.1787654e-1_dp,&
    -0.420059e-2_dp/)
ax=abs(x)
if (ax < 3.75) then          Polynomial fit.
    bessi1_s=ax*poly(real((x/3.75_sp)**2,dp),p)
else
    bessi1_s=(exp(ax)/sqrt(ax))*poly(real(3.75_sp/ax,dp),q)
end if
if (x < 0.0) bessi1_s=-bessi1_s
END FUNCTION bessi1_s
```

```
FUNCTION bessi1_v(x)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi1_v
REAL(SP), DIMENSION(size(x)) :: ax
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(7) :: p = (/0.5_dp,0.87890594_dp,&
    0.51498869_dp,0.15084934_dp,0.2658733e-1_dp,&
    0.301532e-2_dp,0.32411e-3_dp/)
REAL(DP), DIMENSION(9) :: q = (/0.39894228_dp,-0.3988024e-1_dp,&
    -0.362018e-2_dp,0.163801e-2_dp,-0.1031555e-1_dp,&
    0.2282967e-1_dp,-0.2895312e-1_dp,0.1787654e-1_dp,&
    -0.420059e-2_dp/)
ax=abs(x)
mask = (ax < 3.75)
where (mask)
    bessi1_v=ax*poly(real((x/3.75_sp)**2,dp),p,mask)
elsewhere
    y=3.75_sp/ax
    bessi1_v=(exp(ax)/sqrt(ax))*poly(real(y,dp),q,.not. mask)
end where
where (x < 0.0) bessi1_v=-bessi1_v
END FUNCTION bessi1_v
```

⋆ ⋆ ⋆

```
FUNCTION bessk1_s(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessi1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessk1_s
    Returns the modified Bessel function K_1(x) for positive real x.
REAL(DP) :: y                    Accumulate polynomials in double precision.
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,0.15443144_dp,&
    -0.67278579_dp,-0.18156897_dp,-0.1919402e-1_dp,&
    -0.110404e-2_dp,-0.4686e-4_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,0.23498619_dp,&
    -0.3655620e-1_dp,0.1504268e-1_dp,-0.780353e-2_dp,&
    0.325614e-2_dp,-0.68245e-3_dp/)
call assert(x > 0.0, 'bessk1_s arg')
if (x <= 2.0) then               Polynomial fit.
    y=x*x/4.0_sp
    bessk1_s=(log(x/2.0_sp)*bessi1(x))+(1.0_sp/x)*poly(y,p)
else
    y=2.0_sp/x
    bessk1_s=(exp(-x)/sqrt(x))*poly(y,q)
end if
END FUNCTION bessk1_s
```

```
FUNCTION bessk1_v(x)
USE nrtype; USE nrutil, ONLY : assert,poly
USE nr, ONLY : bessi1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessk1_v
REAL(DP), DIMENSION(size(x)) :: y
LOGICAL(LGT), DIMENSION(size(x)) :: mask
REAL(DP), DIMENSION(7) :: p = (/1.0_dp,0.15443144_dp,&
    -0.67278579_dp,-0.18156897_dp,-0.1919402e-1_dp,&
    -0.110404e-2_dp,-0.4686e-4_dp/)
REAL(DP), DIMENSION(7) :: q = (/1.25331414_dp,0.23498619_dp,&
    -0.3655620e-1_dp,0.1504268e-1_dp,-0.780353e-2_dp,&
    0.325614e-2_dp,-0.68245e-3_dp/)
call assert(all(x > 0.0), 'bessk1_v arg')
mask = (x <= 2.0)
where (mask)
    y=x*x/4.0_sp
    bessk1_v=(log(x/2.0_sp)*bessi1(x))+(1.0_sp/x)*poly(y,p,mask)
elsewhere
    y=2.0_sp/x
    bessk1_v=(exp(-x)/sqrt(x))*poly(y,q,.not. mask)
end where
END FUNCTION bessk1_v
```

⋆ ⋆ ⋆

```
FUNCTION bessk_s(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessk0,bessk1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessk_s
    Returns the modified Bessel function K_n(x) for positive x and n ≥ 2.
INTEGER(I4B) :: j
REAL(SP) :: bk,bkm,bkp,tox
call assert(n >= 2, x > 0.0, 'bessk_s args')
tox=2.0_sp/x
bkm=bessk0(x)                Upward recurrence for all x...
bk=bessk1(x)
do j=1,n-1                   ...and here it is.
    bkp=bkm+j*tox*bk
    bkm=bk
    bk=bkp
end do
bessk_s=bk
END FUNCTION bessk_s
```

```
FUNCTION bessk_v(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessk0,bessk1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessk_v
INTEGER(I4B) :: j
REAL(SP), DIMENSION(size(x)) :: bk,bkm,bkp,tox
call assert(n >= 2, all(x > 0.0), 'bessk_v args')
tox=2.0_sp/x
bkm=bessk0(x)
bk=bessk1(x)
do j=1,n-1
    bkp=bkm+j*tox*bk
    bkm=bk
    bk=bkp
end do
bessk_v=bk
END FUNCTION bessk_v
```

The scalar and vector versions of `bessk` are identical, and have no precision-specific constants, another example of where we would like to define a generic "template" function if the language had this facility.

⋆ ⋆ ⋆

```
FUNCTION bessi_s(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessi0
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP) :: bessi_s
INTEGER(I4B), PARAMETER :: IACC=40,IEXP=maxexponent(x)/2
```

Returns the modified Bessel function $I_{\mathtt{n}}(\mathtt{x})$ for any real x and $\mathtt{n} \geq 2$. Make the parameter IACC larger to increase accuracy.

```
INTEGER(I4B) :: j,m
REAL(SP) :: bi,bim,bip,tox
call assert(n >= 2, 'bessi_s args')
bessi_s=0.0
if (x*x <= 8.0_sp*tiny(x)) RETURN          Underflow limit.
tox=2.0_sp/abs(x)
bip=0.0
bi=1.0
m=2*((n+int(sqrt(real(IACC*n,sp)))))       Downward recurrence from even m.
do j=m,1,-1
    bim=bip+j*tox*bi                       The downward recurrence.
    bip=bi
    bi=bim
    if (exponent(bi) > IEXP) then          Renormalize to prevent overflows.
        bessi_s=scale(bessi_s,-IEXP)
        bi=scale(bi,-IEXP)
        bip=scale(bip,-IEXP)
    end if
    if (j == n) bessi_s=bip
end do
bessi_s=bessi_s*bessi0(x)/bi               Normalize with bessi0.
if (x < 0.0 .and. mod(n,2) == 1) bessi_s=-bessi_s
END FUNCTION bessi_s
```

```
if (exponent(bi) > IEXP) then
```

See discussion of scaling for bessj on p. 1107.

```
FUNCTION bessi_v(n,x)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessi0
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: bessi_v
INTEGER(I4B), PARAMETER :: IACC=40,IEXP=maxexponent(x)/2
INTEGER(I4B) :: j,m
REAL(SP), DIMENSION(size(x)) :: bi,bim,bip,tox
LOGICAL(LGT), DIMENSION(size(x)) :: mask
call assert(n >= 2, 'bessi_v args')
bessi_v=0.0
mask = (x <= 8.0_sp*tiny(x))
tox=2.0_sp/merge(2.0_sp,abs(x),mask)
bip=0.0
bi=1.0_sp
m=2*((n+int(sqrt(real(IACC*n,sp)))))
do j=m,1,-1
    bim=bip+j*tox*bi
    bip=bi
    bi=bim
    where (exponent(bi) > IEXP)
        bessi_v=scale(bessi_v,-IEXP)
```

```
        bi=scale(bi,-IEXP)
        bip=scale(bip,-IEXP)
    end where
    if (j == n) bessi_v=bip
end do
bessi_v=bessi_v*bessi0(x)/bi
where (mask) bessi_v=0.0_sp
where (x < 0.0 .and. mod(n,2) == 1) bessi_v=-bessi_v
END FUNCTION bessi_v
```

```
mask = (x == 0.0)
tox=2.0_sp/merge(2.0_sp,abs(x),mask)
```

For the special case $x = 0$, the value of the returned function should be zero; however, the evaluation of `tox` will give a divide check. We substitute an innocuous value for the zero cases, then fix up their answers at the end.

⋆ ⋆ ⋆

```
SUBROUTINE bessjy_s(x,xnu,rj,ry,rjp,ryp)
USE nrtype; USE nrutil, ONLY : assert,nrerror
USE nr, ONLY : beschb
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,xnu
REAL(SP), INTENT(OUT) :: rj,ry,rjp,ryp
INTEGER(I4B), PARAMETER :: MAXIT=10000
REAL(DP), PARAMETER :: XMIN=2.0_dp,EPS=1.0e-10_dp,FPMIN=1.0e-30_dp
```

Returns the Bessel functions $\mathtt{rj} = J_\nu$, $\mathtt{ry} = Y_\nu$ and their derivatives $\mathtt{rjp} = J'_\nu$, $\mathtt{ryp} = Y'_\nu$, for positive `x` and for $\mathtt{xnu} = \nu \geq 0$. The relative accuracy is within one or two significant digits of EPS, except near a zero of one of the functions, where EPS controls its absolute accuracy. FPMIN is a number close to the machine's smallest floating-point number. All internal arithmetic is in double precision. To convert the entire routine to double precision, change the SP declaration above and decrease EPS to 10^{-16}. Also convert the subroutine `beschb`.

```
INTEGER(I4B) :: i,isign,l,nl
REAL(DP) :: a,b,c,d,del,del1,e,f,fact,fact2,fact3,ff,gam,gam1,gam2,&
    gammi,gampl,h,p,pimu,pimu2,q,r,rjl,rjl1,rjmu,rjp1,rjpl,rjtemp,&
    ry1,rymu,rymup,rytemp,sum,sum1,w,x2,xi,xi2,xmu,xmu2
COMPLEX(DPC) :: aa,bb,cc,dd,dl,pq
call assert(x > 0.0, xnu >= 0.0, 'bessjy args')
nl=merge(int(xnu+0.5_dp), max(0,int(xnu-x+1.5_dp)), x < XMIN)
```

`nl` is the number of downward recurrences of the J's and upward recurrences of Y's. `xmu` lies between $-1/2$ and $1/2$ for `x` $<$ `XMIN`, while it is chosen so that `x` is greater than the turning point for `x` $\geq$ `XMIN`.

```
xmu=xnu-nl
xmu2=xmu*xmu
xi=1.0_dp/x
xi2=2.0_dp*xi
w=xi2/PI_D
isign=1
h=xnu*xi
if (h < FPMIN) h=FPMIN
b=xi2*xnu
d=0.0
c=h
do i=1,MAXIT
    b=b+xi2
    d=b-d
    if (abs(d) < FPMIN) d=FPMIN
    c=b-1.0_dp/c
```

The Wronskian.

Evaluate CF1 by modified Lentz's method (§5.2). `isign` keeps track of sign changes in the denominator.

```
    if (abs(c) < FPMIN) c=FPMIN
    d=1.0_dp/d
    del=c*d
    h=del*h
    if (d < 0.0) isign=-isign
    if (abs(del-1.0_dp) < EPS) exit
end do
if (i > MAXIT) call nrerror('x too large in bessjy; try asymptotic expansion')
rjl=isign*FPMIN                          Initialize J_ν and J'_ν for downward recurrence.
rjpl=h*rjl
rjl1=rjl                                 Store values for later rescaling.
rjp1=rjpl
fact=xnu*xi
do l=nl,1,-1
    rjtemp=fact*rjl+rjpl
    fact=fact-xi
    rjpl=fact*rjtemp-rjl
    rjl=rjtemp
end do
if (rjl == 0.0) rjl=EPS
f=rjpl/rjl                               Now have unnormalized J_μ and J'_μ.
if (x < XMIN) then                       Use series.
    x2=0.5_dp*x
    pimu=PI_D*xmu
    if (abs(pimu) < EPS) then
        fact=1.0
    else
        fact=pimu/sin(pimu)
    end if
    d=-log(x2)
    e=xmu*d
    if (abs(e) < EPS) then
        fact2=1.0
    else
        fact2=sinh(e)/e
    end if
    call beschb(xmu,gam1,gam2,gampl,gammi)      Chebyshev evaluation of Γ1 and Γ2.
    ff=2.0_dp/PI_D*fact*(gam1*cosh(e)+gam2*fact2*d)      f_0.
    e=exp(e)
    p=e/(gampl*PI_D)                                     p_0.
    q=1.0_dp/(e*PI_D*gammi)                              q_0.
    pimu2=0.5_dp*pimu
    if (abs(pimu2) < EPS) then
        fact3=1.0
    else
        fact3=sin(pimu2)/pimu2
    end if
    r=PI_D*pimu2*fact3*fact3
    c=1.0
    d=-x2*x2
    sum=ff+r*q
    sum1=p
    do i=1,MAXIT
        ff=(i*ff+p+q)/(i*i-xmu2)
        c=c*d/i
        p=p/(i-xmu)
        q=q/(i+xmu)
        del=c*(ff+r*q)
        sum=sum+del
        del1=c*p-i*del
        sum1=sum1+del1
        if (abs(del) < (1.0_dp+abs(sum))*EPS) exit
    end do
    if (i > MAXIT) call nrerror('bessy series failed to converge')
```

```
    rymu=-sum
    ry1=-sum1*xi2
    rymup=xmu*xi*rymu-ry1
    rjmu=w/(rymup-f*rymu)                    Equation (6.7.13).
else                                         Evaluate CF2 by modified Lentz's method
    a=0.25_dp-xmu2                               (§5.2).
    pq=cmplx(-0.5_dp*xi,1.0_dp,kind=dpc)
    aa=cmplx(0.0_dp,xi*a,kind=dpc)
    bb=cmplx(2.0_dp*x,2.0_dp,kind=dpc)
    cc=bb+aa/pq
    dd=1.0_dp/bb
    pq=cc*dd*pq
    do i=2,MAXIT
        a=a+2*(i-1)
        bb=bb+cmplx(0.0_dp,2.0_dp,kind=dpc)
        dd=a*dd+bb
        if (absc(dd) < FPMIN) dd=FPMIN
        cc=bb+a/cc
        if (absc(cc) < FPMIN) cc=FPMIN
        dd=1.0_dp/dd
        dl=cc*dd
        pq=pq*dl
        if (absc(dl-1.0_dp) < EPS) exit
    end do
    if (i > MAXIT) call nrerror('cf2 failed in bessjy')
    p=real(pq)
    q=aimag(pq)
    gam=(p-f)/q                              Equations (6.7.6) – (6.7.10).
    rjmu=sqrt(w/((p-f)*gam+q))
    rjmu=sign(rjmu,rjl)
    rymu=rjmu*gam
    rymup=rymu*(p+q/gam)
    ry1=xmu*xi*rymu-rymup
end if
fact=rjmu/rjl
rj=rjl1*fact                                 Scale original J_ν and J'_ν.
rjp=rjp1*fact
do i=1,nl                                    Upward recurrence of Y_ν.
    rytemp=(xmu+i)*xi2*ry1-rymu
    rymu=ry1
    ry1=rytemp
end do
ry=rymu
ryp=xnu*xi*rymu-ry1
CONTAINS

FUNCTION absc(z)
IMPLICIT NONE
COMPLEX(DPC), INTENT(IN) :: z
REAL(DP) :: absc
absc=abs(real(z))+abs(aimag(z))
END FUNCTION absc
END SUBROUTINE bessjy_s
```

Yes there is a vector version `bessjy_v`. Its general scheme is to have a bunch of contained functions for various cases, and then combine their outputs (somewhat like `bessj_v`, above, but much more complicated). A listing runs to about four printed pages, and we judge it to be of not much interest, so we will not include it here. (It is included on the machine-readable media.)

⋆ ⋆ ⋆

```
SUBROUTINE beschb_s(x,gam1,gam2,gampl,gammi)
USE nrtype
USE nr, ONLY : chebev
IMPLICIT NONE
REAL(DP), INTENT(IN) :: x
REAL(DP), INTENT(OUT) :: gam1,gam2,gampl,gammi
INTEGER(I4B), PARAMETER :: NUSE1=5,NUSE2=5
```

Evaluates Γ_1 and Γ_2 by Chebyshev expansion for $|\mathtt{x}| \leq 1/2$. Also returns $1/\Gamma(1+\mathtt{x})$ and $1/\Gamma(1-\mathtt{x})$. If converting to double precision, set NUSE1 $= 7$, NUSE2 $= 8$.

```
REAL(SP) :: xx
REAL(SP), DIMENSION(7) :: c1=(/-1.142022680371168_sp,&
    6.5165112670737e-3_sp,3.087090173086e-4_sp,-3.4706269649e-6_sp,&
    6.9437664e-9_sp,3.67795e-11_sp,-1.356e-13_sp/)
REAL(SP), DIMENSION(8) :: c2=(/1.843740587300905_sp,&
    -7.68528408447867e-2_sp,1.2719271366546e-3_sp,&
    -4.9717367042e-6_sp, -3.31261198e-8_sp,2.423096e-10_sp,&
    -1.702e-13_sp,-1.49e-15_sp/)
xx=8.0_dp*x*x-1.0_dp          Multiply x by 2 to make range be −1 to 1, and then apply
gam1=chebev(-1.0_sp,1.0_sp,c1(1:NUSE1),xx)        transformation for evaluating even Cheby-
gam2=chebev(-1.0_sp,1.0_sp,c2(1:NUSE2),xx)        shev series.
gampl=gam2-x*gam1
gammi=gam2+x*gam1
END SUBROUTINE beschb_s
```

```
SUBROUTINE beschb_v(x,gam1,gam2,gampl,gammi)
USE nrtype
USE nr, ONLY : chebev
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(IN) :: x
REAL(DP), DIMENSION(:), INTENT(OUT) :: gam1,gam2,gampl,gammi
INTEGER(I4B), PARAMETER :: NUSE1=5,NUSE2=5
REAL(SP), DIMENSION(size(x)) :: xx
REAL(SP), DIMENSION(7) :: c1=(/-1.142022680371168_sp,&
    6.5165112670737e-3_sp,3.087090173086e-4_sp,-3.4706269649e-6_sp,&
    6.9437664e-9_sp,3.67795e-11_sp,-1.356e-13_sp/)
REAL(SP), DIMENSION(8) :: c2=(/1.843740587300905_sp,&
    -7.68528408447867e-2_sp,1.2719271366546e-3_sp,&
    -4.9717367042e-6_sp, -3.31261198e-8_sp,2.423096e-10_sp,&
    -1.702e-13_sp,-1.49e-15_sp/)
xx=8.0_dp*x*x-1.0_dp
gam1=chebev(-1.0_sp,1.0_sp,c1(1:NUSE1),xx)
gam2=chebev(-1.0_sp,1.0_sp,c2(1:NUSE2),xx)
gampl=gam2-x*gam1
gammi=gam2+x*gam1
END SUBROUTINE beschb_v
```

⋆ ⋆ ⋆

```
SUBROUTINE bessik(x,xnu,ri,rk,rip,rkp)
USE nrtype; USE nrutil, ONLY : assert,nrerror
USE nr, ONLY : beschb
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,xnu
REAL(SP), INTENT(OUT) :: ri,rk,rip,rkp
INTEGER(I4B), PARAMETER :: MAXIT=10000
REAL(SP), PARAMETER :: XMIN=2.0
REAL(DP), PARAMETER :: EPS=1.0e-10_dp,FPMIN=1.0e-30_dp
```

Returns the modified Bessel functions ri $= I_\nu$, rk $= K_\nu$ and their derivatives rip $= I'_\nu$, rkp $= K'_\nu$, for positive x and for xnu $= \nu \geq 0$. The relative accuracy is within one or

two significant digits of EPS. FPMIN is a number close to the machine's smallest floating-point number. All internal arithmetic is in double precision. To convert the entire routine to double precision, change the REAL declaration above and decrease EPS to 10^{-16}. Also convert the subroutine beschb.

```
INTEGER(I4B) :: i,l,nl
REAL(DP) :: a,a1,b,c,d,del,del1,delh,dels,e,f,fact,fact2,ff,&
    gam1,gam2,gammi,gampl,h,p,pimu,q,q1,q2,qnew,&
    ril,ril1,rimu,rip1,ripl,ritemp,rk1,rkmu,rkmup,rktemp,&
    s,sum,sum1,x2,xi,xi2,xmu,xmu2
call assert(x > 0.0, xnu >= 0.0, 'bessik args')
nl=int(xnu+0.5_dp)                          nl is the number of downward recurrences
xmu=xnu-nl                                      of the I's and upward recurrences
xmu2=xmu*xmu                                    of K's. xmu lies between −1/2 and
xi=1.0_dp/x                                     1/2.
xi2=2.0_dp*xi
h=xnu*xi                                    Evaluate CF1 by modified Lentz's method
if (h < FPMIN) h=FPMIN                          (§5.2).
b=xi2*xnu
d=0.0
c=h
do i=1,MAXIT
    b=b+xi2
    d=1.0_dp/(b+d)                          Denominators cannot be zero here, so no
    c=b+1.0_dp/c                                need for special precautions.
    del=c*d
    h=del*h
    if (abs(del-1.0_dp) < EPS) exit
end do
if (i > MAXIT) call nrerror('x too large in bessik; try asymptotic expansion')
ril=FPMIN                                   Initialize I_ν and I'_ν for downward recur-
ripl=h*ril                                      rence.
ril1=ril                                    Store values for later rescaling.
rip1=ripl
fact=xnu*xi
do l=nl,1,-1
    ritemp=fact*ril+ripl
    fact=fact-xi
    ripl=fact*ritemp+ril
    ril=ritemp
end do
f=ripl/ril                                  Now have unnormalized I_μ and I'_μ.
if (x < XMIN) then                          Use series.
    x2=0.5_dp*x
    pimu=PI_D*xmu
    if (abs(pimu) < EPS) then
        fact=1.0
    else
        fact=pimu/sin(pimu)
    end if
    d=-log(x2)
    e=xmu*d
    if (abs(e) < EPS) then
        fact2=1.0
    else
        fact2=sinh(e)/e
    end if
    call beschb(xmu,gam1,gam2,gampl,gammi)      Chebyshev evaluation of Γ1 and Γ2.
    ff=fact*(gam1*cosh(e)+gam2*fact2*d)         f_0.
    sum=ff
    e=exp(e)
    p=0.5_dp*e/gampl                            p_0.
    q=0.5_dp/(e*gammi)                          q_0.
    c=1.0
    d=x2*x2
```

```
    sum1=p
    do i=1,MAXIT
        ff=(i*ff+p+q)/(i*i-xmu2)
        c=c*d/i
        p=p/(i-xmu)
        q=q/(i+xmu)
        del=c*ff
        sum=sum+del
        del1=c*(p-i*ff)
        sum1=sum1+del1
        if (abs(del) < abs(sum)*EPS) exit
    end do
    if (i > MAXIT) call nrerror('bessk series failed to converge')
    rkmu=sum
    rk1=sum1*xi2
else                                        Evaluate CF2 by Steed's algorithm (§5.2),
    b=2.0_dp*(1.0_dp+x)                         which is OK because there can be no
    d=1.0_dp/b                                  zero denominators.
    delh=d
    h=delh
    q1=0.0                                  Initializations for recurrence (6.7.35).
    q2=1.0
    a1=0.25_dp-xmu2
    c=a1
    q=c                                     First term in equation (6.7.34).
    a=-a1
    s=1.0_dp+q*delh
    do i=2,MAXIT
        a=a-2*(i-1)
        c=-a*c/i
        qnew=(q1-b*q2)/a
        q1=q2
        q2=qnew
        q=q+c*qnew
        b=b+2.0_dp
        d=1.0_dp/(b+a*d)
        delh=(b*d-1.0_dp)*delh
        h=h+delh
        dels=q*delh
        s=s+dels
        if (abs(dels/s) < EPS) exit         Need only test convergence of sum, since
    end do                                      CF2 itself converges more quickly.
    if (i > MAXIT) call nrerror('bessik: failure to converge in cf2')
    h=a1*h
    rkmu=sqrt(PI_D/(2.0_dp*x))*exp(-x)/s    Omit the factor exp(−x) to scale all the
    rk1=rkmu*(xmu+x+0.5_dp-h)*xi                returned functions by exp(x) for x ≥
end if                                          XMIN.
rkmup=xmu*xi*rkmu-rk1
rimu=xi/(f*rkmu-rkmup)                      Get I_μ from Wronskian.
ri=(rimu*ril1)/ril                          Scale original I_ν and I'_ν.
rip=(rimu*rip1)/ril
do i=1,nl                                   Upward recurrence of K_ν.
    rktemp=(xmu+i)*xi2*rk1+rkmu
    rkmu=rk1
    rk1=rktemp
end do
rk=rkmu
rkp=xnu*xi*rkmu-rk1
END SUBROUTINE bessik
```

bessik does not readily parallelize, and we thus don't provide a vector version. Since airy, immediately following, requires bessik, we don't have a vector version of it, either.

⋆ ⋆ ⋆

```
SUBROUTINE airy(x,ai,bi,aip,bip)
USE nrtype
USE nr, ONLY : bessik,bessjy
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: ai,bi,aip,bip
    Returns Airy functions Ai(x), Bi(x), and their derivatives Ai'(x), Bi'(x).
REAL(SP) :: absx,ri,rip,rj,rjp,rk,rkp,rootx,ry,ryp,z
REAL(SP), PARAMETER :: THIRD=1.0_sp/3.0_sp,TWOTHR=2.0_sp/3.0_sp, &
    ONOVRT=0.5773502691896258_sp
absx=abs(x)
rootx=sqrt(absx)
z=TWOTHR*absx*rootx
if (x > 0.0) then
    call bessik(z,THIRD,ri,rk,rip,rkp)
    ai=rootx*ONOVRT*rk/PI
    bi=rootx*(rk/PI+2.0_sp*ONOVRT*ri)
    call bessik(z,TWOTHR,ri,rk,rip,rkp)
    aip=-x*ONOVRT*rk/PI
    bip=x*(rk/PI+2.0_sp*ONOVRT*ri)
else if (x < 0.0) then
    call bessjy(z,THIRD,rj,ry,rjp,ryp)
    ai=0.5_sp*rootx*(rj-ONOVRT*ry)
    bi=-0.5_sp*rootx*(ry+ONOVRT*rj)
    call bessjy(z,TWOTHR,rj,ry,rjp,ryp)
    aip=0.5_sp*absx*(ONOVRT*ry+rj)
    bip=0.5_sp*absx*(ONOVRT*rj-ry)
else                                Case x = 0.
    ai=0.3550280538878172_sp
    bi=ai/ONOVRT
    aip=-0.2588194037928068_sp
    bip=-aip/ONOVRT
end if
END SUBROUTINE airy
```

⋆ ⋆ ⋆

```
SUBROUTINE sphbes_s(n,x,sj,sy,sjp,syp)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessjy
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: sj,sy,sjp,syp
    Returns spherical Bessel functions j_n(x), y_n(x), and their derivatives j'_n(x), y'_n(x) for
    integer n ≥ 0 and x > 0.
REAL(SP), PARAMETER :: RTPIO2=1.253314137315500_sp
REAL(SP) :: factor,order,rj,rjp,ry,ryp
call assert(n >= 0, x > 0.0, 'sphbes_s args')
order=n+0.5_sp
call bessjy(x,order,rj,ry,rjp,ryp)
factor=RTPIO2/sqrt(x)
sj=factor*rj
sy=factor*ry
```

```
sjp=factor*rjp-sj/(2.0_sp*x)
syp=factor*ryp-sy/(2.0_sp*x)
END SUBROUTINE sphbes_s


SUBROUTINE sphbes_v(n,x,sj,sy,sjp,syp)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : bessjy
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(OUT) :: sj,sy,sjp,syp
REAL(SP), PARAMETER :: RTPIO2=1.253314137315500_sp
REAL(SP) :: order
REAL(SP), DIMENSION(size(x)) :: factor,rj,rjp,ry,ryp
call assert(n >= 0, all(x > 0.0), 'sphbes_v args')
order=n+0.5_sp
call bessjy(x,order,rj,ry,rjp,ryp)
factor=RTPIO2/sqrt(x)
sj=factor*rj
sy=factor*ry
sjp=factor*rjp-sj/(2.0_sp*x)
syp=factor*ryp-sy/(2.0_sp*x)
END SUBROUTINE sphbes_v
```

Note that `sphbes_v` uses (through overloading) `bessjy_v`. The listing of that routine was omitted above, but it is on the machine-readable media.

⋆ ⋆ ⋆

```
FUNCTION plgndr_s(l,m,x)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: l,m
REAL(SP), INTENT(IN) :: x
REAL(SP) :: plgndr_s
```

Computes the associated Legendre polynomial $P_l^m(x)$. Here m and l are integers satisfying $0 \leq m \leq l$, while x lies in the range $-1 \leq x \leq 1$.

```
INTEGER(I4B) :: ll
REAL(SP) :: pll,pmm,pmmp1,somx2
call assert(m >= 0, m <= l, abs(x) <= 1.0, 'plgndr_s args')
pmm=1.0                          Compute P_m^m.
if (m > 0) then
    somx2=sqrt((1.0_sp-x)*(1.0_sp+x))
    pmm=product(arth(1.0_sp,2.0_sp,m))*somx2**m
    if (mod(m,2) == 1) pmm=-pmm
end if
if (l == m) then
    plgndr_s=pmm
else
    pmmp1=x*(2*m+1)*pmm          Compute P_{m+1}^m.
    if (l == m+1) then
        plgndr_s=pmmp1
    else                         Compute P_l^m, l > m + 1.
        do ll=m+2,l
            pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m)
            pmm=pmmp1
            pmmp1=pll
        end do
        plgndr_s=pll
```

```
    end if
end if
END FUNCTION plgndr_s
```

```
product(arth(1.0_sp,2.0_sp,m))
```

That is, $(2m-1)!!$

```
FUNCTION plgndr_v(l,m,x)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: l,m
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: plgndr_v
INTEGER(I4B) :: ll
REAL(SP), DIMENSION(size(x)) :: pll,pmm,pmmp1,somx2
call assert(m >= 0, m <= l, all(abs(x) <= 1.0), 'plgndr_v args')
pmm=1.0
if (m > 0) then
    somx2=sqrt((1.0_sp-x)*(1.0_sp+x))
    pmm=product(arth(1.0_sp,2.0_sp,m))*somx2**m
    if (mod(m,2) == 1) pmm=-pmm
end if
if (l == m) then
    plgndr_v=pmm
else
    pmmp1=x*(2*m+1)*pmm
    if (l == m+1) then
        plgndr_v=pmmp1
    else
        do ll=m+2,l
            pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m)
            pmm=pmmp1
            pmmp1=pll
        end do
        plgndr_v=pll
    end if
end if
END FUNCTION plgndr_v
```

All those `if`'s (not `where`'s) may strike you as odd in a vector routine, but it is vectorized only on x, the dependent variable, not on the scalar indices l and m. Much harder to write a routine that is parallel for a vector of arbitrary triplets (l, m, x). Try it!

⋆ ⋆ ⋆

```
SUBROUTINE frenel(x,s,c)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: s,c
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=tiny(x),BIG=huge(x)*EPS,&
    XMIN=1.5
```

Computes the Fresnel integrals $S(x)$ and $C(x)$ for all real x.

Parameters: `MAXIT` is the maximum number of iterations allowed; `EPS` is the relative error; `FPMIN` is a number near the smallest representable floating-point number; `BIG` is a number

near the machine overflow limit; XMIN is the dividing line between using the series and continued fraction.

```
INTEGER(I4B) :: k,n
REAL(SP) :: a,ax,fact,pix2,sign,sum,sumc,sums,term,test
COMPLEX(SPC) :: b,cc,d,h,del,cs
LOGICAL(LGT) :: odd
ax=abs(x)
if (ax < sqrt(FPMIN)) then          Special case: avoid failure of convergence test be-
    s=0.0                               cause of underflow.
    c=ax
else if (ax <= XMIN) then           Evaluate both series simultaneously.
    sum=0.0
    sums=0.0
    sumc=ax
    sign=1.0
    fact=PIO2*ax*ax
    odd=.true.
    term=ax
    n=3
    do k=1,MAXIT
        term=term*fact/k
        sum=sum+sign*term/n
        test=abs(sum)*EPS
        if (odd) then
            sign=-sign
            sums=sum
            sum=sumc
        else
            sumc=sum
            sum=sums
        end if
        if (term < test) exit
        odd=.not. odd
        n=n+2
    end do
    if (k > MAXIT) call nrerror('frenel: series failed')
    s=sums
    c=sumc
else                                Evaluate continued fraction by modified Lentz's method
    pix2=PI*ax*ax                       (§5.2).
    b=cmplx(1.0_sp,-pix2,kind=spc)
    cc=BIG
    d=1.0_sp/b
    h=d
    n=-1
    do k=2,MAXIT
        n=n+2
        a=-n*(n+1)
        b=b+4.0_sp
        d=1.0_sp/(a*d+b)            Denominators cannot be zero.
        cc=b+a/cc
        del=cc*d
        h=h*del
        if (absc(del-1.0_sp) <= EPS) exit
    end do
    if (k > MAXIT) call nrerror('cf failed in frenel')
    h=h*cmplx(ax,-ax,kind=spc)
    cs=cmplx(0.5_sp,0.5_sp,kind=spc)*(1.0_sp-&
        cmplx(cos(0.5_sp*pix2),sin(0.5_sp*pix2),kind=spc)*h)
    c=real(cs)
    s=aimag(cs)
end if
if (x < 0.0) then                   Use antisymmetry.
    c=-c
```

```
    s=-s
end if
CONTAINS
FUNCTION absc(z)
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: z
REAL(SP) :: absc
absc=abs(real(z))+abs(aimag(z))
END FUNCTION absc
END SUBROUTINE frenel
```

`b=cmplx(1.0_sp,-pix2,kind=spc)` It's a good idea *always* to include the `kind=` parameter when you use the `cmplx` intrinsic. The reason is that, perhaps counterintuitively, the result of `cmplx` is not determined by the kind of its arguments, but is rather the "default complex kind." Since that default may not be what you think it is (or what `spc` is defined to be), the desired kind should be specified explicitly.

`c=real(cs)` And why not specify a `kind=` parameter here, where it is also optionally allowed? Our answer is that the `real` intrinsic actually merges two different usages. When its argument is complex, it is the counterpart of `aimag` and returns a value whose kind is determined by the kind of its argument. In fact `aimag` doesn't even allow an optional kind parameter, so we never put one in the corresponding use of `real`. The other usage of `real` is for "casting," that is, converting one real type to another (e.g., double precision to single precision, or vice versa). Here we *always* include a kind parameter, since otherwise the result is the default real kind, with the same dangers mentioned in the previous paragraph.

⋆ ⋆ ⋆

```
SUBROUTINE cisi(x,ci,si)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), INTENT(OUT) :: ci,si
INTEGER(I4B), PARAMETER :: MAXIT=100
REAL(SP), PARAMETER :: EPS=epsilon(x),FPMIN=4.0_sp*tiny(x),&
    BIG=huge(x)*EPS,TMIN=2.0
```

Computes the cosine and sine integrals $\mathrm{Ci}(x)$ and $\mathrm{Si}(x)$. $\mathrm{Ci}(0)$ is returned as a large negative number and no error message is generated. For $x < 0$ the routine returns $\mathrm{Ci}(-x)$ and you must supply the $-i\pi$ yourself.

Parameters: `MAXIT` is the maximum number of iterations allowed; `EPS` is the relative error, or absolute error near a zero of $\mathrm{Ci}(x)$; `FPMIN` is a number near the smallest representable floating-point number; `BIG` is a number near the machine overflow limit; `TMIN` is the dividing line between using the series and continued fraction; `EULER` $= \gamma$ (in `nrtype`).

```
INTEGER(I4B) :: i,k
REAL(SP) :: a,err,fact,sign,sum,sumc,sums,t,term
COMPLEX(SPC) :: h,b,c,d,del
LOGICAL(LGT) :: odd
t=abs(x)
if (t == 0.0) then                          Special case.
    si=0.0
    ci=-BIG
    RETURN
end if                                      Evaluate continued fraction by modified Lentz's
if (t > TMIN) then                              method (§5.2).
    b=cmplx(1.0_sp,t,kind=spc)
```

```
    c=BIG
    d=1.0_sp/b
    h=d
    do i=2,MAXIT
        a=-(i-1)**2
        b=b+2.0_sp
        d=1.0_sp/(a*d+b)                Denominators cannot be zero.
        c=b+a/c
        del=c*d
        h=h*del
        if (absc(del-1.0_sp) <= EPS) exit
    end do
    if (i > MAXIT) call nrerror('continued fraction failed in cisi')
    h=cmplx(cos(t),-sin(t),kind=spc)*h
    ci=-real(h)
    si=PIO2+aimag(h)
else                                    Evaluate both series simultaneously.
    if (t < sqrt(FPMIN)) then           Special case: avoid failure of convergence test
        sumc=0.0                            because of underflow.
        sums=t
    else
        sum=0.0
        sums=0.0
        sumc=0.0
        sign=1.0
        fact=1.0
        odd=.true.
        do k=1,MAXIT
            fact=fact*t/k
            term=fact/k
            sum=sum+sign*term
            err=term/abs(sum)
            if (odd) then
                sign=-sign
                sums=sum
                sum=sumc
            else
                sumc=sum
                sum=sums
            end if
            if (err < EPS) exit
            odd=.not. odd
        end do
        if (k > MAXIT) call nrerror('MAXIT exceeded in cisi')
    end if
    si=sums
    ci=sumc+log(t)+EULER
end if
if (x < 0.0) si=-si
CONTAINS

FUNCTION absc(z)
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: z
REAL(SP) :: absc
absc=abs(real(z))+abs(aimag(z))
END FUNCTION absc
END SUBROUTINE cisi
```

⋆ ⋆ ⋆

```
FUNCTION dawson_s(x)
USE nrtype; USE nrutil, ONLY : arth,geop
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: dawson_s
    Returns Dawson's integral F(x) = exp(-x^2) int_0^x exp(t^2)dt for any real x.
INTEGER(I4B), PARAMETER :: NMAX=6
REAL(SP), PARAMETER :: H=0.4_sp,A1=2.0_sp/3.0_sp,A2=0.4_sp,&
    A3=2.0_sp/7.0_sp
INTEGER(I4B) :: i,n0
REAL(SP) :: ec,x2,xp,xx
REAL(SP), DIMENSION(NMAX) :: d1,d2,e1
REAL(SP), DIMENSION(NMAX), SAVE :: c=(/ (0.0_sp,i=1,NMAX) /)
if (c(1) == 0.0) c(1:NMAX)=exp(-(arth(1,2,NMAX)*H)**2)
  Initialize c on first call.
if (abs(x) < 0.2_sp) then                    Use series expansion.
    x2=x**2
    dawson_s=x*(1.0_sp-A1*x2*(1.0_sp-A2*x2*(1.0_sp-A3*x2)))
else                                         Use sampling theorem representation.
    xx=abs(x)
    n0=2*nint(0.5_sp*xx/H)
    xp=xx-real(n0,sp)*H
    ec=exp(2.0_sp*xp*H)
    d1=arth(n0+1,2,NMAX)
    d2=arth(n0-1,-2,NMAX)
    e1=geop(ec,ec**2,NMAX)
    dawson_s=0.5641895835477563_sp*sign(exp(-xp**2),x)*&    Constant is 1/sqrt(pi).
        sum(c*(e1/d1+1.0_sp/(d2*e1)))
end if
END FUNCTION dawson_s
```

f90 `REAL(SP), DIMENSION(NMAX), SAVE :: c=(/ (0.0_sp,i=1,NMAX) /)` This is one way to give initial values to an array. Actually, we're somewhat nervous about using the "implied do-loop" form of the array constructor, as above, because our parallel compilers might not always be smart enough to execute the constructor in parallel. In this case, with `NMAX=6`, the damage potential is quite minimal. An alternative way to initialize the array would be with a data statement, "`DATA c /NMAX*0.0_sp/`"; however, this is not considered good Fortran 90 style, and there is no reason to think that it would be faster.

`c(1:NMAX)=exp(-(arth(1,2,NMAX)*H)**2)` Another example where the `arth` function of `nrutil` comes in handy. Otherwise, this would be

```
do i=1,NMAX
   c(i)=exp(-((2.0_sp*i-1.0_sp)*H)**2)
end do
```

`arth(n0+1,2,NMAX)...arth(n0-1,-2,NMAX)...geop(ec,ec**2,NMAX)` These are not just notationally convenient for generating the sequences $(n_0+1, n_0+3, n_0+5, \ldots)$, $(n_0-1, n_0-3, n_0-5, \ldots)$, and $(\mathtt{ec}, \mathtt{ec}^3, \mathtt{ec}^5, \ldots)$. They also may allow parallelization with parallel versions of `arth` and `geop`, such as those in `nrutil`.

```
FUNCTION dawson_v(x)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP), DIMENSION(size(x)) :: dawson_v
INTEGER(I4B), PARAMETER :: NMAX=6
```

```
REAL(SP), PARAMETER :: H=0.4_sp,A1=2.0_sp/3.0_sp,A2=0.4_sp,&
    A3=2.0_sp/7.0_sp
INTEGER(I4B) :: i,n
REAL(SP), DIMENSION(size(x)) :: x2
REAL(SP), DIMENSION(NMAX), SAVE :: c=(/ (0.0_sp,i=1,NMAX) /)
LOGICAL(LGT), DIMENSION(size(x)) :: mask
if (c(1) == 0.0) c(1:NMAX)=exp(-(arth(1,2,NMAX)*H)**2)
mask = (abs(x) >= 0.2_sp)
dawson_v=dawsonseries_v(x,mask)
where (.not. mask)
    x2=x**2
    dawson_v=x*(1.0_sp-A1*x2*(1.0_sp-A2*x2*(1.0_sp-A3*x2)))
end where
CONTAINS
FUNCTION dawsonseries_v(xin,mask)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xin
LOGICAL(LGT), DIMENSION(size(xin)), INTENT(IN) :: mask
REAL(SP), DIMENSION(size(xin)) :: dawsonseries_v
INTEGER(I4B), DIMENSION(:), ALLOCATABLE :: n0
REAL(SP), DIMENSION(:), ALLOCATABLE :: d1,d2,e1,e2,sm,xp,xx,x
n=count(mask)
if (n == 0) RETURN
allocate(n0(n),d1(n),d2(n),e1(n),e2(n),sm(n),xp(n),xx(n),x(n))
x=pack(xin,mask)
xx=abs(x)
n0=2*nint(0.5_sp*xx/H)
xp=xx-real(n0,sp)*H
e1=exp(2.0_sp*xp*H)
e2=e1**2
d1=n0+1.0_sp
d2=d1-2.0_sp
sm=0.0
do i=1,NMAX
    sm=sm+c(i)*(e1/d1+1.0_sp/(d2*e1))
    d1=d1+2.0_sp
    d2=d2-2.0_sp
    e1=e2*e1
end do
sm=0.5641895835477563_sp*sign(exp(-xp**2),x)*sm
dawsonseries_v=unpack(sm,mask,0.0_sp)
deallocate(n0,d1,d2,e1,e2,sm,xp,xx)
END FUNCTION dawsonseries_v
END FUNCTION dawson_v
```

`dawson_v=dawsonseries_v(x,mask)` Pass-the-buck method for getting masked values, see note to `bessj0_v` above, p. 1102. Within the contained `dawsonseries`, we use the pack-unpack method. Note that, unlike in `dawson_s`, the sums are done by do-loops, because the parallelization is already over the components of the vector argument.

⋆ ⋆ ⋆

```
FUNCTION rf_s(x,y,z)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y,z
REAL(SP) :: rf_s
REAL(SP), PARAMETER :: ERRTOL=0.08_sp,TINY=1.5e-38_sp,BIG=3.0e37_sp,&
    THIRD=1.0_sp/3.0_sp,&
```

```
    C1=1.0_sp/24.0_sp,C2=0.1_sp,C3=3.0_sp/44.0_sp,C4=1.0_sp/14.0_sp
```

Computes Carlson's elliptic integral of the first kind, $R_F(x, y, z)$. x, y, and z must be nonnegative, and at most one can be zero. TINY must be at least 5 times the machine underflow limit, BIG at most one-fifth the machine overflow limit.

```
REAL(SP) :: alamb,ave,delx,dely,delz,e2,e3,sqrtx,sqrty,sqrtz,xt,yt,zt
call assert(min(x,y,z) >= 0.0, min(x+y,x+z,y+z) >= TINY, &
    max(x,y,z) <= BIG, 'rf_s args')
xt=x
yt=y
zt=z
do
    sqrtx=sqrt(xt)
    sqrty=sqrt(yt)
    sqrtz=sqrt(zt)
    alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
    xt=0.25_sp*(xt+alamb)
    yt=0.25_sp*(yt+alamb)
    zt=0.25_sp*(zt+alamb)
    ave=THIRD*(xt+yt+zt)
    delx=(ave-xt)/ave
    dely=(ave-yt)/ave
    delz=(ave-zt)/ave
    if (max(abs(delx),abs(dely),abs(delz)) <= ERRTOL) exit
end do
e2=delx*dely-delz**2
e3=delx*dely*delz
rf_s=(1.0_sp+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave)
END FUNCTION rf_s



FUNCTION rf_v(x,y,z)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
REAL(SP), DIMENSION(size(x)) :: rf_v
REAL(SP), PARAMETER :: ERRTOL=0.08_sp,TINY=1.5e-38_sp,BIG=3.0e37_sp,&
    THIRD=1.0_sp/3.0_sp,&
    C1=1.0_sp/24.0_sp,C2=0.1_sp,C3=3.0_sp/44.0_sp,C4=1.0_sp/14.0_sp
REAL(SP), DIMENSION(size(x)) :: alamb,ave,delx,dely,delz,e2,e3,&
    sqrtx,sqrty,sqrtz,xt,yt,zt
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(I4B) :: ndum
ndum=assert_eq(size(x),size(y),size(z),'rf_v')
call assert(all(min(x,y,z) >= 0.0), all(min(x+y,x+z,y+z) >= TINY), &
    all(max(x,y,z) <= BIG), 'rf_v args')
xt=x
yt=y
zt=z
converged=.false.
do
    where (.not. converged)
        sqrtx=sqrt(xt)
        sqrty=sqrt(yt)
        sqrtz=sqrt(zt)
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
        xt=0.25_sp*(xt+alamb)
        yt=0.25_sp*(yt+alamb)
        zt=0.25_sp*(zt+alamb)
        ave=THIRD*(xt+yt+zt)
        delx=(ave-xt)/ave
        dely=(ave-yt)/ave
        delz=(ave-zt)/ave
        converged = (max(abs(delx),abs(dely),abs(delz)) <= ERRTOL)
```

```
    end where
    if (all(converged)) exit
end do
e2=delx*dely-delz**2
e3=delx*dely*delz
rf_v=(1.0_sp+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave)
END FUNCTION rf_v
```

```
FUNCTION rd_s(x,y,z)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y,z
REAL(SP) :: rd_s
REAL(SP), PARAMETER :: ERRTOL=0.05_sp,TINY=1.0e-25_sp,BIG=4.5e21_sp,&
    C1=3.0_sp/14.0_sp,C2=1.0_sp/6.0_sp,C3=9.0_sp/22.0_sp,&
    C4=3.0_sp/26.0_sp,C5=0.25_sp*C3,C6=1.5_sp*C4
```

Computes Carlson's elliptic integral of the second kind, $R_D(x, y, z)$. x and y must be nonnegative, and at most one can be zero. z must be positive. TINY must be at least twice the negative $2/3$ power of the machine overflow limit. BIG must be at most $0.1 \times$ ERRTOL times the negative $2/3$ power of the machine underflow limit.

```
REAL(SP) :: alamb,ave,delx,dely,delz,ea,eb,ec,ed,&
    ee,fac,sqrtx,sqrty,sqrtz,sum,xt,yt,zt
call assert(min(x,y) >= 0.0, min(x+y,z) >= TINY, max(x,y,z) <= BIG, &
    'rd_s args')
xt=x
yt=y
zt=z
sum=0.0
fac=1.0
do
    sqrtx=sqrt(xt)
    sqrty=sqrt(yt)
    sqrtz=sqrt(zt)
    alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
    sum=sum+fac/(sqrtz*(zt+alamb))
    fac=0.25_sp*fac
    xt=0.25_sp*(xt+alamb)
    yt=0.25_sp*(yt+alamb)
    zt=0.25_sp*(zt+alamb)
    ave=0.2_sp*(xt+yt+3.0_sp*zt)
    delx=(ave-xt)/ave
    dely=(ave-yt)/ave
    delz=(ave-zt)/ave
    if (max(abs(delx),abs(dely),abs(delz)) <= ERRTOL) exit
end do
ea=delx*dely
eb=delz*delz
ec=ea-eb
ed=ea-6.0_sp*eb
ee=ed+ec+ec
rd_s=3.0_sp*sum+fac*(1.0_sp+ed*(-C1+C5*ed-C6*delz*ee)&
    +delz*(C2*ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave))
END FUNCTION rd_s
```

```
FUNCTION rd_v(x,y,z)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
REAL(SP), DIMENSION(size(x)) :: rd_v
REAL(SP), PARAMETER :: ERRTOL=0.05_sp,TINY=1.0e-25_sp,BIG=4.5e21_sp,&
    C1=3.0_sp/14.0_sp,C2=1.0_sp/6.0_sp,C3=9.0_sp/22.0_sp,&
    C4=3.0_sp/26.0_sp,C5=0.25_sp*C3,C6=1.5_sp*C4
REAL(SP), DIMENSION(size(x)) :: alamb,ave,delx,dely,delz,ea,eb,ec,ed,&
    ee,fac,sqrtx,sqrty,sqrtz,sum,xt,yt,zt
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(I4B) :: ndum
ndum=assert_eq(size(x),size(y),size(z),'rd_v')
call assert(all(min(x,y) >= 0.0), all(min(x+y,z) >= TINY), &
    all(max(x,y,z) <= BIG), 'rd_v args')
xt=x
yt=y
zt=z
sum=0.0
fac=1.0
converged=.false.
do
    where (.not. converged)
        sqrtx=sqrt(xt)
        sqrty=sqrt(yt)
        sqrtz=sqrt(zt)
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
        sum=sum+fac/(sqrtz*(zt+alamb))
        fac=0.25_sp*fac
        xt=0.25_sp*(xt+alamb)
        yt=0.25_sp*(yt+alamb)
        zt=0.25_sp*(zt+alamb)
        ave=0.2_sp*(xt+yt+3.0_sp*zt)
        delx=(ave-xt)/ave
        dely=(ave-yt)/ave
        delz=(ave-zt)/ave
        converged = (all(max(abs(delx),abs(dely),abs(delz)) <= ERRTOL))
    end where
    if (all(converged)) exit
end do
ea=delx*dely
eb=delz*delz
ec=ea-eb
ed=ea-6.0_sp*eb
ee=ed+ec+ec
rd_v=3.0_sp*sum+fac*(1.0_sp+ed*(-C1+C5*ed-C6*delz*ee)&
    +delz*(C2*ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave))
END FUNCTION rd_v
```

```
FUNCTION rj_s(x,y,z,p)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : rc,rf
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y,z,p
REAL(SP) :: rj_s
REAL(SP), PARAMETER :: ERRTOL=0.05_sp,TINY=2.5e-13_sp,BIG=9.0e11_sp,&
    C1=3.0_sp/14.0_sp,C2=1.0_sp/3.0_sp,C3=3.0_sp/22.0_sp,&
    C4=3.0_sp/26.0_sp,C5=0.75_sp*C3,C6=1.5_sp*C4,C7=0.5_sp*C2,&
    C8=C3+C3
```

Computes Carlson's elliptic integral of the third kind, $R_J(x, y, z, p)$. x, y, and z must be nonnegative, and at most one can be zero. p must be nonzero. If $p < 0$, the Cauchy

principal value is returned. TINY must be at least twice the cube root of the machine underflow limit, BIG at most one-fifth the cube root of the machine overflow limit.

```
REAL(SP) :: a,alamb,alpha,ave,b,bet,delp,delx,&
    dely,delz,ea,eb,ec,ed,ee,fac,pt,rho,sqrtx,sqrty,sqrtz,&
    sm,tau,xt,yt,zt
call assert(min(x,y,z) >= 0.0, min(x+y,x+z,y+z,abs(p)) >= TINY, &
    max(x,y,z,abs(p)) <= BIG, 'rj_s args')
sm=0.0
fac=1.0
if (p > 0.0) then
    xt=x
    yt=y
    zt=z
    pt=p
else
    xt=min(x,y,z)
    zt=max(x,y,z)
    yt=x+y+z-xt-zt
    a=1.0_sp/(yt-p)
    b=a*(zt-yt)*(yt-xt)
    pt=yt+b
    rho=xt*zt/yt
    tau=p*pt/yt
end if
do
    sqrtx=sqrt(xt)
    sqrty=sqrt(yt)
    sqrtz=sqrt(zt)
    alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
    alpha=(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz)**2
    bet=pt*(pt+alamb)**2
    sm=sm+fac*rc(alpha,bet)
    fac=0.25_sp*fac
    xt=0.25_sp*(xt+alamb)
    yt=0.25_sp*(yt+alamb)
    zt=0.25_sp*(zt+alamb)
    pt=0.25_sp*(pt+alamb)
    ave=0.2_sp*(xt+yt+zt+pt+pt)
    delx=(ave-xt)/ave
    dely=(ave-yt)/ave
    delz=(ave-zt)/ave
    delp=(ave-pt)/ave
    if (max(abs(delx),abs(dely),abs(delz),abs(delp)) <= ERRTOL) exit
end do
ea=delx*(dely+delz)+dely*delz
eb=delx*dely*delz
ec=delp**2
ed=ea-3.0_sp*ec
ee=eb+2.0_sp*delp*(ea-ec)
rj_s=3.0_sp*sm+fac*(1.0_sp+ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8&
    +delp*C4))+delp*ea*(C2-delp*C3)-C2*delp*ec)/(ave*sqrt(ave))
if (p <= 0.0) rj_s=a*(b*rj_s+3.0_sp*(rc(rho,tau)-rf(xt,yt,zt)))
END FUNCTION rj_s



FUNCTION rj_v(x,y,z,p)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : rc,rf
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z,p
REAL(SP), DIMENSION(size(x)) :: rj_v
REAL(SP), PARAMETER :: ERRTOL=0.05_sp,TINY=2.5e-13_sp,BIG=9.0e11_sp,&
    C1=3.0_sp/14.0_sp,C2=1.0_sp/3.0_sp,C3=3.0_sp/22.0_sp,&
```

```
    C4=3.0_sp/26.0_sp,C5=0.75_sp*C3,C6=1.5_sp*C4,C7=0.5_sp*C2,&
    C8=C3+C3
REAL(SP), DIMENSION(size(x)) :: a,alamb,alpha,ave,b,bet,delp,delx,&
    dely,delz,ea,eb,ec,ed,ee,fac,pt,rho,sqrtx,sqrty,sqrtz,&
    sm,tau,xt,yt,zt
LOGICAL(LGT), DIMENSION(size(x)) :: mask
INTEGER(I4B) :: ndum
ndum=assert_eq(size(x),size(y),size(z),size(p),'rj_v')
call assert(all(min(x,y,z) >= 0.0), all(min(x+y,x+z,y+z,abs(p)) >= TINY), &
    all(max(x,y,z,abs(p)) <= BIG), 'rj_v args')
sm=0.0
fac=1.0
where (p > 0.0)
    xt=x
    yt=y
    zt=z
    pt=p
elsewhere
    xt=min(x,y,z)
    zt=max(x,y,z)
    yt=x+y+z-xt-zt
    a=1.0_sp/(yt-p)
    b=a*(zt-yt)*(yt-xt)
    pt=yt+b
    rho=xt*zt/yt
    tau=p*pt/yt
end where
mask=.false.
do
    where (.not. mask)
        sqrtx=sqrt(xt)
        sqrty=sqrt(yt)
        sqrtz=sqrt(zt)
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz
        alpha=(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz)**2
        bet=pt*(pt+alamb)**2
        sm=sm+fac*rc(alpha,bet)
        fac=0.25_sp*fac
        xt=0.25_sp*(xt+alamb)
        yt=0.25_sp*(yt+alamb)
        zt=0.25_sp*(zt+alamb)
        pt=0.25_sp*(pt+alamb)
        ave=0.2_sp*(xt+yt+zt+pt+pt)
        delx=(ave-xt)/ave
        dely=(ave-yt)/ave
        delz=(ave-zt)/ave
        delp=(ave-pt)/ave
        mask = (max(abs(delx),abs(dely),abs(delz),abs(delp)) <= ERRTOL)
    end where
    if (all(mask)) exit
end do
ea=delx*(dely+delz)+dely*delz
eb=delx*dely*delz
ec=delp**2
ed=ea-3.0_sp*ec
ee=eb+2.0_sp*delp*(ea-ec)
rj_v=3.0_sp*sm+fac*(1.0_sp+ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8&
    +delp*C4))+delp*ea*(C2-delp*C3)-C2*delp*ec)/(ave*sqrt(ave))
mask = (p <= 0.0)
where (mask) rj_v=a*(b*rj_v+&
    unpack(3.0_sp*(rc(pack(rho,mask),pack(tau,mask))-&
    rf(pack(xt,mask),pack(yt,mask),pack(zt,mask))),mask,0.0_sp))
END FUNCTION rj_v
```

f90 `unpack(3.0_sp*(rc(pack(rho,mask),pack(tau,mask))...),mask,0.0_sp)`
If you're willing to put up with fairly unreadable code, you can use the pack-unpack trick (for getting a masked subset of components out of a vector function) right in-line, as here. Of course the "outer level" that is seen by the enclosing `where` construction has to contain only objects that have the same shape as the `mask` that goes with the `where`. Because it is so hard to read, we don't like to do this very often. An alternative would be to use `CONTAINS` to incorporate short, masked "wrapper functions" for the functions used in this way.

```
FUNCTION rc_s(x,y)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP) :: rc_s
REAL(SP), PARAMETER :: ERRTOL=0.04_sp,TINY=1.69e-38_sp,&
    SQRTNY=1.3e-19_sp,BIG=3.0e37_sp,TNBG=TINY*BIG,&
    COMP1=2.236_sp/SQRTNY,COMP2=TNBG*TNBG/25.0_sp,&
    THIRD=1.0_sp/3.0_sp,&
    C1=0.3_sp,C2=1.0_sp/7.0_sp,C3=0.375_sp,C4=9.0_sp/22.0_sp
```

Computes Carlson's degenerate elliptic integral, $R_C(x, y)$. x must be nonnegative and y must be nonzero. If $y < 0$, the Cauchy principal value is returned. `TINY` must be at least 5 times the machine underflow limit, `BIG` at most one-fifth the machine maximum overflow limit.

```
REAL(SP) :: alamb,ave,s,w,xt,yt
call assert( (/x >= 0.0,y /= 0.0,x+abs(y) >= TINY,x+abs(y) <= BIG, &
    y >= -COMP1 .or. x <= 0.0 .or. x >= COMP2/),'rc_s')
if (y > 0.0) then
    xt=x
    yt=y
    w=1.0
else
    xt=x-y
    yt=-y
    w=sqrt(x)/sqrt(xt)
end if
do
    alamb=2.0_sp*sqrt(xt)*sqrt(yt)+yt
    xt=0.25_sp*(xt+alamb)
    yt=0.25_sp*(yt+alamb)
    ave=THIRD*(xt+yt+yt)
    s=(yt-ave)/ave
    if (abs(s) <= ERRTOL) exit
end do
rc_s=w*(1.0_sp+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave)
END FUNCTION rc_s
```

```
FUNCTION rc_v(x,y)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(size(x)) :: rc_v
REAL(SP), PARAMETER :: ERRTOL=0.04_sp,TINY=1.69e-38_sp,&
    SQRTNY=1.3e-19_sp,BIG=3.0e37_sp,TNBG=TINY*BIG,&
    COMP1=2.236_sp/SQRTNY,COMP2=TNBG*TNBG/25.0_sp,&
    THIRD=1.0_sp/3.0_sp,&
    C1=0.3_sp,C2=1.0_sp/7.0_sp,C3=0.375_sp,C4=9.0_sp/22.0_sp
REAL(SP), DIMENSION(size(x)) :: alamb,ave,s,w,xt,yt
LOGICAL(LGT), DIMENSION(size(x)) :: converged
INTEGER(I4B) :: ndum
```

```
ndum=assert_eq(size(x),size(y),'rc_v')
call assert( (/all(x >= 0.0),all(y /= 0.0),all(x+abs(y) >= TINY), &
    all(x+abs(y) <= BIG),all(y >= -COMP1 .or. x <= 0.0 &
    .or. x >= COMP2) /),'rc_v')
where (y > 0.0)
    xt=x
    yt=y
    w=1.0
elsewhere
    xt=x-y
    yt=-y
    w=sqrt(x)/sqrt(xt)
end where
converged=.false.
do
    where (.not. converged)
        alamb=2.0_sp*sqrt(xt)*sqrt(yt)+yt
        xt=0.25_sp*(xt+alamb)
        yt=0.25_sp*(yt+alamb)
        ave=THIRD*(xt+yt+yt)
        s=(yt-ave)/ave
        converged = (abs(s) <= ERRTOL)
    end where
    if (all(converged)) exit
end do
rc_v=w*(1.0_sp+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave)
END FUNCTION rc_v
```

⋆ ⋆ ⋆

```
FUNCTION ellf_s(phi,ak)
USE nrtype
USE nr, ONLY : rf
IMPLICIT NONE
REAL(SP), INTENT(IN) :: phi,ak
REAL(SP) :: ellf_s
```

Legendre elliptic integral of the 1st kind $F(\phi, k)$, evaluated using Carlson's function R_F. The argument ranges are $0 \leq \phi \leq \pi/2$, $0 \leq k \sin\phi \leq 1$.

```
REAL(SP) :: s
s=sin(phi)
ellf_s=s*rf(cos(phi)**2,(1.0_sp-s*ak)*(1.0_sp+s*ak),1.0_sp)
END FUNCTION ellf_s
```

```
FUNCTION ellf_v(phi,ak)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : rf
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
REAL(SP), DIMENSION(size(phi)) :: ellf_v
REAL(SP), DIMENSION(size(phi)) :: s
INTEGER(I4B) :: ndum
ndum=assert_eq(size(phi),size(ak),'ellf_v')
s=sin(phi)
ellf_v=s*rf(cos(phi)**2,(1.0_sp-s*ak)*(1.0_sp+s*ak),&
    spread(1.0_sp,1,size(phi)))
END FUNCTION ellf_v
```

```
FUNCTION elle_s(phi,ak)
USE nrtype
USE nr, ONLY : rd,rf
IMPLICIT NONE
REAL(SP), INTENT(IN) :: phi,ak
REAL(SP) :: elle_s
```

Legendre elliptic integral of the 2nd kind $E(\phi,k)$, evaluated using Carlson's functions R_D and R_F. The argument ranges are $0 \leq \phi \leq \pi/2$, $0 \leq k \sin\phi \leq 1$.

```
REAL(SP) :: cc,q,s
s=sin(phi)
cc=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
elle_s=s*(rf(cc,q,1.0_sp)-((s*ak)**2)*rd(cc,q,1.0_sp)/3.0_sp)
END FUNCTION elle_s
```

```
FUNCTION elle_v(phi,ak)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : rd,rf
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
REAL(SP), DIMENSION(size(phi)) :: elle_v
REAL(SP), DIMENSION(size(phi)) :: cc,q,s
INTEGER(I4B) :: ndum
ndum=assert_eq(size(phi),size(ak),'elle_v')
s=sin(phi)
cc=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
elle_v=s*(rf(cc,q,spread(1.0_sp,1,size(phi)))-((s*ak)**2)*&
    rd(cc,q,spread(1.0_sp,1,size(phi)))/3.0_sp)
END FUNCTION elle_v
```

f90 `rd(cc,q,spread(1.0_sp,1,size(phi)))` See note to `erf_v`, p. 1094 above.

```
FUNCTION ellpi_s(phi,en,ak)
USE nrtype
USE nr, ONLY : rf,rj
IMPLICIT NONE
REAL(SP), INTENT(IN) :: phi,en,ak
REAL(SP) :: ellpi_s
```

Legendre elliptic integral of the 3rd kind $\Pi(\phi,n,k)$, evaluated using Carlson's functions R_J and R_F. (Note that the sign convention on n is opposite that of Abramowitz and Stegun.) The ranges of ϕ and k are $0 \leq \phi \leq \pi/2$, $0 \leq k \sin\phi \leq 1$.

```
REAL(SP) :: cc,enss,q,s
s=sin(phi)
enss=en*s*s
cc=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
ellpi_s=s*(rf(cc,q,1.0_sp)-enss*rj(cc,q,1.0_sp,1.0_sp+enss)/3.0_sp)
END FUNCTION ellpi_s
```

```
FUNCTION ellpi_v(phi,en,ak)
USE nrtype
USE nr, ONLY : rf,rj
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: phi,en,ak
REAL(SP), DIMENSION(size(phi)) :: ellpi_v
REAL(SP), DIMENSION(size(phi)) :: cc,enss,q,s
s=sin(phi)
enss=en*s*s
cc=cos(phi)**2
q=(1.0_sp-s*ak)*(1.0_sp+s*ak)
ellpi_v=s*(rf(cc,q,spread(1.0_sp,1,size(phi)))-enss*&
    rj(cc,q,spread(1.0_sp,1,size(phi)),1.0_sp+enss)/3.0_sp)
END FUNCTION ellpi_v
```

⋆ ⋆ ⋆

```
SUBROUTINE sncndn(uu,emmc,sn,cn,dn)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: uu,emmc
REAL(SP), INTENT(OUT) :: sn,cn,dn
```

Returns the Jacobian elliptic functions sn(u, k_c), cn(u, k_c), and dn(u, k_c). Here uu $= u$, while emmc $= k_c^2$.

```
REAL(SP), PARAMETER :: CA=0.0003_sp
```

The accuracy is the square of CA.

```
INTEGER(I4B), PARAMETER :: MAXIT=13
INTEGER(I4B) :: i,ii,l
REAL(SP) :: a,b,c,d,emc,u
REAL(SP), DIMENSION(MAXIT) :: em,en
LOGICAL(LGT) :: bo
emc=emmc
u=uu
if (emc /= 0.0) then
    bo=(emc < 0.0)
    if (bo) then
        d=1.0_sp-emc
        emc=-emc/d
        d=sqrt(d)
        u=d*u
    end if
    a=1.0
    dn=1.0
    do i=1,MAXIT
        l=i
        em(i)=a
        emc=sqrt(emc)
        en(i)=emc
        c=0.5_sp*(a+emc)
        if (abs(a-emc) <= CA*a) exit
        emc=a*emc
        a=c
    end do
    if (i > MAXIT) call nrerror('sncndn: convergence failed')
    u=c*u
    sn=sin(u)
    cn=cos(u)
    if (sn /= 0.0) then
        a=cn/sn
        c=a*c
        do ii=l,1,-1
            b=em(ii)
```

```
            a=c*a
            c=dn*c
            dn=(en(ii)+a)/(b+a)
            a=c/b
        end do
        a=1.0_sp/sqrt(c**2+1.0_sp)
        sn=sign(a,sn)
        cn=c*sn
    end if
    if (bo) then
        a=dn
        dn=cn
        cn=a
        sn=sn/d
    end if
else
    cn=1.0_sp/cosh(u)
    dn=cn
    sn=tanh(u)
end if
END SUBROUTINE sncndn
```

⋆ ⋆ ⋆

```
MODULE hypgeo_info
USE nrtype
COMPLEX(SPC) :: hypgeo_aa,hypgeo_bb,hypgeo_cc,hypgeo_dz,hypgeo_z0
END MODULE hypgeo_info
```

```
FUNCTION hypgeo(a,b,c,z)
USE nrtype
USE hypgeo_info
USE nr, ONLY : bsstep,hypdrv,hypser,odeint
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: a,b,c,z
COMPLEX(SPC) :: hypgeo
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Complex hypergeometric function ${}_2F_1$ for complex a, b, c, and z, by direct integration of the hypergeometric equation in the complex plane. The branch cut is taken to lie along the real axis, Re $z > 1$.

Parameter: EPS is an accuracy parameter.

```
COMPLEX(SPC), DIMENSION(2) :: y
REAL(SP), DIMENSION(4) :: ry
if (real(z)**2+aimag(z)**2 <= 0.25) then      Use series...
    call hypser(a,b,c,z,hypgeo,y(2))
    RETURN
else if (real(z) < 0.0) then                  ...or pick a starting point for the path
    hypgeo_z0=cmplx(-0.5_sp,0.0_sp,kind=spc)      integration.
else if (real(z) <= 1.0) then
    hypgeo_z0=cmplx(0.5_sp,0.0_sp,kind=spc)
else
    hypgeo_z0=cmplx(0.0_sp,sign(0.5_sp,aimag(z)),kind=spc)
end if
hypgeo_aa=a                                   Load the module variables, used to pass
hypgeo_bb=b                                       parameters "over the head" of odeint
hypgeo_cc=c                                       to hypdrv.
hypgeo_dz=z-hypgeo_z0
call hypser(hypgeo_aa,hypgeo_bb,hypgeo_cc,hypgeo_z0,y(1),y(2))
```

Get starting function and derivative.

```
ry(1:4:2)=real(y)
```

```
ry(2:4:2)=aimag(y)
call odeint(ry,0.0_sp,1.0_sp,EPS,0.1_sp,0.0001_sp,hypdrv,bsstep)
```
The arguments to odeint are the vector of independent variables, the starting and ending values of the dependent variable, the accuracy parameter, an initial guess for stepsize, a minimum stepsize, and the names of the derivative routine and the (here Bulirsch-Stoer) stepping routine.
```
y=cmplx(ry(1:4:2),ry(2:4:2),kind=spc)
hypgeo=y(1)
END FUNCTION hypgeo



SUBROUTINE hypser(a,b,c,z,series,deriv)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: a,b,c,z
COMPLEX(SPC), INTENT(OUT) :: series,deriv
```
Returns the hypergeometric series ${}_2F_1$ and its derivative, iterating to machine accuracy. For cabs(z) $\leq 1/2$ convergence is quite rapid.
```
INTEGER(I4B) :: n
INTEGER(I4B), PARAMETER :: MAXIT=1000
COMPLEX(SPC) :: aa,bb,cc,fac,temp
deriv=cmplx(0.0_sp,0.0_sp,kind=spc)
fac=cmplx(1.0_sp,0.0_sp,kind=spc)
temp=fac
aa=a
bb=b
cc=c
do n=1,MAXIT
    fac=((aa*bb)/cc)*fac
    deriv=deriv+fac
    fac=fac*z/n
    series=temp+fac
    if (series == temp) RETURN
    temp=series
    aa=aa+1.0
    bb=bb+1.0
    cc=cc+1.0
end do
call nrerror('hypser: convergence failure')
END SUBROUTINE hypser



SUBROUTINE hypdrv(s,ry,rdyds)
USE nrtype
USE hypgeo_info
IMPLICIT NONE
REAL(SP), INTENT(IN) :: s
REAL(SP), DIMENSION(:), INTENT(IN) :: ry
REAL(SP), DIMENSION(:), INTENT(OUT) :: rdyds
```
Derivative subroutine for the hypergeometric equation; see text equation (5.14.4).
```
COMPLEX(SPC), DIMENSION(2) :: y,dyds
COMPLEX(SPC) :: z
y=cmplx(ry(1:4:2),ry(2:4:2),kind=spc)
z=hypgeo_z0+s*hypgeo_dz
dyds(1)=y(2)*hypgeo_dz
dyds(2)=((hypgeo_aa*hypgeo_bb)*y(1)-(hypgeo_cc-&
    ((hypgeo_aa+hypgeo_bb)+1.0_sp)*z)*y(2))*hypgeo_dz/(z*(1.0_sp-z))
rdyds(1:4:2)=real(dyds)
rdyds(2:4:2)=aimag(dyds)
END SUBROUTINE hypdrv
```

f90 Notice that the real array (of length 4) `ry` is immediately mapped into a complex array of length 2, and that the process is reversed at the end of the routine with `rdyds`. In Fortran 77 no such mapping is necessary: the calling program sends real arguments, and the Fortran 77 `hypdrv` simply interprets what is sent as complex. Fortran 90's stronger typing does not encourage (and, practically, does not allow) this convenience; but it is a small price to pay for the vastly increased error-checking capabilities of a strongly typed language.

Chapter B7. Random Numbers

One might think that good random number generators, including those in Volume 1, should last forever. The world of computing changes very rapidly, however:

- When Volume 1 was published, it was unusual, except on the fastest supercomputers, to "exhaust" a 32-bit random number generator, that is, to call for all 2^{32} sequential random values in its periodic sequence. Now, this is feasible, and not uncommon, on fast desktop workstations. A useful generator today must have a minimum of 64 bits of state space, and generally somewhat more.
- Before Fortran 90, the Fortran language had no standardized calling sequence for random numbers. Now, although there is still no standard *algorithm* defined by the language (rightly, we think), there is at least a standard calling sequence, exemplified in the intrinsics `random_number` and `random_seed`.
- The rise of parallel computing places new algorithmic demands on random generators. The classic algorithms, which compute each random value from the previous one, evidently need generalization to a parallel environment.
- New algorithms and techniques have been discovered, in some cases significantly faster than their predecessors.

These are the reasons that we have decided to implement, in Fortran 90, different uniform random number generators from those in Volume 1's Fortran 77 implementations. We hasten to add that there is nothing wrong with any of the generators in Volume 1. That volume's `ran0` and `ran1` routines are, to our knowledge, completely adequate as 32-bit generators; `ran2` has a 64-bit state space, and our previous offer of $1000 for *any* demonstrated failure in the algorithm has never yet been claimed (see [1]).

Before we launch into the discussion of parallelizable generators with Fortran 90 calling conventions, we want to attend to the continuing needs of longtime "`x=ran(idum)`" users with purely serial machines. If you are a satisfied user of Volume 1's `ran0`, `ran1`, or `ran2` Fortran 77 versions, you are in this group. The following routine, `ran`, preserves those routines' calling conventions, is considerably faster than `ran2`, and does not suffer from the old `ran0` or `ran1`'s 32-bit period exhaustion limitation. It is completely portable to all Fortran 90 environments. We recommend `ran` as the plug-compatible replacement for the old `ran0`, `ran1`, and `ran2`, and we happily offer exactly the same $1000 reward terms as were (and are still) offered on the old `ran2`.

```
FUNCTION ran(idum)
IMPLICIT NONE
INTEGER, PARAMETER :: K4B=selected_int_kind(9)
INTEGER(K4B), INTENT(INOUT) :: idum
REAL :: ran
```

"Minimal" random number generator of Park and Miller combined with a Marsaglia shift sequence. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This fully portable, scalar generator has the "traditional" (not Fortran 90) calling sequence with a random deviate as the returned function value: call with `idum` a negative integer to initialize; thereafter, do not alter `idum` except to reinitialize. The period of this generator is about 3.1×10^{18}.

```
INTEGER(K4B), PARAMETER :: IA=16807,IM=2147483647,IQ=127773,IR=2836
REAL, SAVE :: am
INTEGER(K4B), SAVE :: ix=-1,iy=-1,k
if (idum <= 0 .or. iy < 0) then          Initialize.
    am=nearest(1.0,-1.0)/IM
    iy=ior(ieor(888889999,abs(idum)),1)
    ix=ieor(777755555,abs(idum))
    idum=abs(idum)+1                     Set idum positive.
end if
ix=ieor(ix,ishft(ix,13))                 Marsaglia shift sequence with period 2^32 - 1.
ix=ieor(ix,ishft(ix,-17))
ix=ieor(ix,ishft(ix,5))
k=iy/IQ                                  Park-Miller sequence by Schrage's method,
iy=IA*(iy-k*IQ)-IR*k                         period 2^31 - 2.
if (iy < 0) iy=iy+IM
ran=am*ior(iand(IM,ieor(ix,iy)),1)       Combine the two generators with masking to
END FUNCTION ran                             ensure nonzero value.
```

This is a good place to discuss a new bit of algorithmics that has crept into `ran`, above, and even more strongly affects all of our new random number generators, below. Consider:

```
ix=ieor(ix,ishft(ix,13))
ix=ieor(ix,ishft(ix,-17))
ix=ieor(ix,ishft(ix,5))
```

These lines update a 32-bit integer `ix`, which cycles pseudo-randomly through a full period of $2^{32} - 1$ values (excluding zero) before repeating. Generators of this type have been extensively explored by Marsaglia (see [2]), who has kindly communicated some additional results to us in advance of publication. For convenience, we will refer to generators of this sort as "Marsaglia shift registers."

Useful properties of Marsaglia shift registers are (i) they are very fast on most machines, since they use only fast logical operations, and (ii) the bit-mixing that they induce is quite different in character from that induced by arithmetic operations such as are used in linear congruential generators (see Volume 1) or lagged Fibonacci generators (see below). Thus, the combination of a Marsaglia shift register with another, algorithmically quite different generator is a powerful way to suppress any residual correlations or other weaknesses in the other generator. Indeed, Marsaglia finds (and we concur) that the above generator (with constants $13, -17, 5$, as shown) is *by itself* about as good as any 32-bit random generator.

Here is a very brief outline of the theory behind these generators: Consider the 32 bits of the integer as components in a vector of length 32, in a linear space where addition and multiplication are done modulo 2. Noting that exclusive-or (`ieor`) is the same as addition, each of the three lines in the updating can be written as the action of a 32×32 matrix on a vector, where the matrix is all zeros except for

ones on the diagonal, and on exactly one super- or subdiagonal (corresponding to positive or negative second arguments in `ishft`). Denote this matrix as $\mathbf{S}_k$, where k is the shift argument. Then, one full step of updating (three lines of code, above) corresponds to multiplication by the matrix $\mathbf{T} \equiv \mathbf{S}_{k_3}\mathbf{S}_{k_2}\mathbf{S}_{k_1}$.

One next needs to find triples of integers (k_1, k_2, k_3), for example $(13, -17, 5)$, that give the full $M \equiv 2^{32} - 1$ period. Necessary and sufficient conditions are that $\mathbf{T}^M = \mathbf{1}$ (the identity matrix), and that $\mathbf{T}^N \neq \mathbf{1}$ for these five values of N: $N = 3 \times 5 \times 17 \times 257$, $N = 3 \times 5 \times 17 \times 65537$, $N = 3 \times 5 \times 257 \times 65537$, $N = 3 \times 17 \times 257 \times 65537$, $N = 5 \times 17 \times 257 \times 65537$. (Note that each of the five prime factors of M is omitted one at a time to get the five values of N.) The required large powers of $\mathbf{T}$ are readily computed by successive squarings, requiring only on the order of $32^3 \log M$ operations. With this machinery, one can find full-period triples (k_1, k_2, k_3) by exhaustive search, at reasonable cost.

Not all such triples are equally good as generators of random integers, however. Marsaglia subjects candidate values to a battery of tests for randomness, and we have ourselves applied various tests. This stage of winnowing is as much art as science, because all 32-bit generators can be made to exhibit signs of failure due to period exhaustion (if for no other reason). "Good" triples, in order of our preference, are $(13, -17, 5)$, $(5, -13, 6)$, $(5, -9, 7)$, $(13, -17, 15)$, $(16, -7, 11)$. When a full-period triple is good, its reverse is also full-period, and also generally good. A good *quadruple* due to Marsaglia (generalizing the above in the obvious way) is $(-4, 8, -1, 5)$. We would not recommend relying on any single Marsaglia shift generator (nor on any other simple generator) *by itself*. Two or more generators, of quite different types, should be combined [1].

⋆ ⋆ ⋆

Let us now discuss explicitly the needs of *parallel* random number generators. The general scheme, from the user's perspective, is that of Fortran 90's intrinsic `random_number`: A statement like `call ran1(harvest)` (where `ran1` will be one of our portable replacements for the compiler-dependent `random_number`) should fill the real array `harvest` with pseudo-random real values in the range $(0, 1)$. Of course, we want the underlying machinery to be completely parallel, that is, no do-loops of order $N \equiv$ `size(harvest)`.

A first design decision is whether to replicate the state-space across the parallel dimension N, i.e., whether to reserve storage for essentially N scalar generators. Although there are various schemes that avoid doing this (e.g., mapping a single, smaller, state space into N different output values on each call), we think that it is a memory cost well worth paying in return for achieving a less exotic (and thus better tested) algorithm. However, this choice dictates that we must keep the state space *per component* quite small. We have settled on five or fewer 32-bit words of state space per component as a reasonable limit. Some otherwise interesting and well tested methods (such as Knuth's subtractive generator, implemented in Volume 1 as `ran3`) are ruled out by this constraint.

A second design decision is how to initialize the parallel state space, so that different parallel components produce different sequences, and so that there is an acceptable degree of randomness *across* the parallel dimension, as well as *between successive calls* of the generator. Each component starts its life with one and only one unique identifier, its component index n in the range $1 \ldots N$. One is

tempted simply to hash the values n into the corresponding components of initial state space. "Random" hashing is a bad idea, however, because different n's will produce identical 32-bit hash results by chance when N is no larger than $\sim 2^{16}$. We therefore prefer to use a kind of reversible pseudo-encryption (similar to the routine psdes in Volume 1 and below) which guarantees causally that different n's produce different state space initializations.

f90 The machinery for allocating, deallocating, and initializing the state space, including provision of a user interface for getting or putting the contents of the state space (as in the intrinsic random_seed) is fairly complicated. Rather than duplicate it in each different random generator that we provide, we have consolidated it in a single module, ran_state, whose contents we will now discuss. Such a discussion is necessarily technical, if not arcane; on first reading, you may wish to skip ahead to the actual new routines ran0, ran1, and ran2. If you do so, you will need to know only that ran_state provides each vector random routine with five 32-bit vectors of state information, denoted iran, jran, kran, mran, nran. (The overloaded scalar generators have five corresponding 32-bit scalars, denoted iran0, etc.)

MODULE ran_state

This module supports the random number routines ran0, ran1, ran2, and ran3. It provides each generator with five integers (for vector versions, five vectors of integers), for use as internal state space. The first three integers (iran, jran, kran) are maintained as nonnegative values, while the last two (mran, nran) have 32-bit nonzero values. Also provided by this module is support for initializing or reinitializing the state space to a desired standard sequence number, hashing the initial values to random values, and allocating and deallocating the internal workspace.

```
USE nrtype
IMPLICIT NONE
INTEGER, PARAMETER :: K4B=selected_int_kind(9)
```

Independent of the usual integer kind I4B, we need a kind value for (ideally) 32-bit integers.

```
INTEGER(K4B), PARAMETER :: hg=huge(1_K4B), hgm=-hg, hgng=hgm-1
INTEGER(K4B), SAVE :: lenran=0, seq=0
INTEGER(K4B), SAVE :: iran0,jran0,kran0,nran0,mran0,rans
INTEGER(K4B), DIMENSION(:,:), POINTER, SAVE :: ranseeds
INTEGER(K4B), DIMENSION(:), POINTER, SAVE :: iran,jran,kran, &
    nran,mran,ranv
REAL(SP), SAVE :: amm
INTERFACE ran_hash
```

Scalar and vector versions of the hashing procedure.

```
    MODULE PROCEDURE ran_hash_s, ran_hash_v
END INTERFACE
```

CONTAINS

(We here intersperse discussion with the listing of the module.) The module defines K4B as an integer KIND that is intended to be 32 bits. If your machine doesn't have 32-bit integers (hard to believe!) this will be caught later, and an error message generated. The definition of the parameters hg, hgm, and hgng makes an assumption about 32-bit integers that goes beyond the strict Fortran 90 integer model, that the magnitude of the most negative representable integer is greater by one than that of the most positive representable integer. This is a property of the *two's complement arithmetic* that is used on virtually all modern machines (see, e.g., [3]).

The global variables rans (for scalar) and ranv (for vector) are used by all of our routines to store the *integer* value associated with the most recently returned call. You can access these (with a "USE ran_state" statement) if you want integer, rather than real, random deviates.

The first routine, ran_init, is called by routines later in the chapter to initialize their state space. It is *not* intended to be called from a user's program.

```
SUBROUTINE ran_init(length)
USE nrtype; USE nrutil, ONLY : arth,nrerror,reallocate
IMPLICIT NONE
INTEGER(K4B), INTENT(IN) :: length
```

Initialize or reinitialize the random generator state space to vectors of size length. The saved variable seq is hashed (via calls to the module routine ran_hash) to create unique starting seeds, different for each vector component.

```
INTEGER(K4B) :: new,j,hgt
if (length < lenran) RETURN                Simply return if enough space is already al-
hgt=hg                                         located.
```

The following lines check that kind value K4B is in fact a 32-bit integer with the usual properties that we expect it to have (under negation and wrap-around addition). If all of these tests are satisfied, then the routines that use this module are portable, even though they go beyond Fortran 90's integer model.

```
if (hg /= 2147483647) call nrerror('ran_init: arith assump 1 fails')
if (hgng >= 0) call nrerror('ran_init: arith assump 2 fails')
if (hgt+1 /= hgng) call nrerror('ran_init: arith assump 3 fails')
if (not(hg) >= 0) call nrerror('ran_init: arith assump 4 fails')
if (not(hgng) < 0) call nrerror('ran_init: arith assump 5 fails')
if (hg+hgng >= 0) call nrerror('ran_init: arith assump 6 fails')
if (not(-1_k4b) < 0) call nrerror('ran_init: arith assump 7 fails')
if (not(0_k4b) >= 0) call nrerror('ran_init: arith assump 8 fails')
if (not(1_k4b) >= 0) call nrerror('ran_init: arith assump 9 fails')
if (lenran > 0) then                       Reallocate space, or ...
    ranseeds=>reallocate(ranseeds,length,5)
    ranv=>reallocate(ranv,length-1)
    new=lenran+1
else                                       allocate space.
    allocate(ranseeds(length,5))
    allocate(ranv(length-1))
    new=1                                  Index of first location not yet initialized.
    amm=nearest(1.0_sp,-1.0_sp)/hgng
```

Use of nearest is to ensure that returned random deviates are strictly less than 1.0.

```
    if (amm*hgng >= 1.0 .or. amm*hgng <= 0.0) &
        call nrerror('ran_init: arth assump 10 fails')
end if
```

Set starting values, unique by seq and vector component.

```
ranseeds(new:,1)=seq
ranseeds(new:,2:5)=spread(arth(new,1,size(ranseeds(new:,1))),2,4)
do j=1,4                                   Hash them.
    call ran_hash(ranseeds(new:,j),ranseeds(new:,j+1))
end do
where (ranseeds(new:,1:3) < 0) &           Enforce nonnegativity.
    ranseeds(new:,1:3)=not(ranseeds(new:,1:3))
where (ranseeds(new:,4:5) == 0) ranseeds(new:,4:5)=1    Enforce nonzero.
if (new == 1) then                         Set scalar seeds.
    iran0=ranseeds(1,1)
    jran0=ranseeds(1,2)
    kran0=ranseeds(1,3)
    mran0=ranseeds(1,4)
    nran0=ranseeds(1,5)
    rans=nran0
end if
if (length > 1) then                       Point to vector seeds.
    iran => ranseeds(2:,1)
    jran => ranseeds(2:,2)
    kran => ranseeds(2:,3)
    mran => ranseeds(2:,4)
    nran => ranseeds(2:,5)
    ranv = nran
```

```
end if
lenran=length
END SUBROUTINE ran_init
```

f90 `hgt=hg ... if (hgt+1 /= hgng)` Bit of dirty laundry here! We are testing whether the most positive integer `hg` wraps around to the most negative integer `hgng` when 1 is added to it. We can't just write `hg+1`, since some compilers will evaluate this at compile time and return an overflow error message. If your compiler sees through the charade of the temporary variable `hgt`, you'll have to find another way to trick it.

`amm=nearest(1.0_sp,-1.0_sp)/hgng...` Logically, `amm` should be a parameter; but the `nearest` intrinsic is trouble-prone in the initialization expression for a parameter (named constant), so we compute this at run time. We then check that `amm`, when multiplied by the largest possible negative integer, does not equal or exceed unity. (Our random deviates are guaranteed never to equal zero or unity exactly.)

You might wonder why `amm` is negative, and why we multiply it by negative integers to get positive random deviates. The answer, which will become manifest in the random generators given below, is that we want to use the fast `not` operation on integers to convert them to nonzero values of all one sign. This is possible if the conversion is to negative values, since `not(i)` is negative for all nonnegative `i`. If the conversion were to positive values, we would have problems both with zero (its sign bit is already positive) and `hgng` (since `not(hgng)` is generally zero).

```
iran0=ranseeds(1,1) ...
iran => ranseeds(2:,1)...
```

The initial state information is stored in `ranseeds`, a two-dimensional array whose column (second) index ranges from 1 to 5 over the state variables. `ranseeds(1,:)` is reserved for scalar random generators, while `ranseeds(2:,:)` is for vector-parallel generators. The `ranseeds` array is made available to vector generators through the pointers `iran`, `jran`, `kran`, `mran`, and `nran`. The corresponding scalar values, `iran0`,. . ., `nran0` are simply global variables, not pointers, because the overhead of addressing a scalar through a pointer is often too great. (We will have to copy these scalar values back into `ranseeds` when it, rarely, needs to be addressed as an array.)

`call ran_hash(...)` Unique, and random, initial state information is obtained by putting a user-settable "sequence number" into `iran`, a component number into `jran`, and hashing this pair. Then `jran` and `kran` are hashed, `kran` and `mran` are hashed, and so forth.

```
SUBROUTINE ran_deallocate
   User interface to release the workspace used by the random number routines.
if (lenran > 0) then
    deallocate(ranseeds,ranv)
    nullify(ranseeds,ranv,iran,jran,kran,mran,nran)
    lenran = 0
end if
END SUBROUTINE ran_deallocate
```

The above routine is supplied as a user interface for deallocating all the state space storage.

```
SUBROUTINE ran_seed(sequence,size,put,get)
IMPLICIT NONE
INTEGER, OPTIONAL, INTENT(IN) :: sequence
INTEGER, OPTIONAL, INTENT(OUT) :: size
INTEGER, DIMENSION(:), OPTIONAL, INTENT(IN) :: put
INTEGER, DIMENSION(:), OPTIONAL, INTENT(OUT) :: get
```

User interface for seeding the random number routines. Syntax is exactly like Fortran 90's random_seed routine, with one additional argument keyword: sequence, set to any integer value, causes an immediate new initialization, seeded by that integer.

```
if (present(size)) then
    size=5*lenran
else if (present(put)) then
    if (lenran == 0) RETURN
    ranseeds=reshape(put,shape(ranseeds))
    where (ranseeds(:,1:3) < 0) ranseeds(:,1:3)=not(ranseeds(:,1:3))
```

Enforce nonnegativity and nonzero conditions on any user-supplied seeds.

```
    where (ranseeds(:,4:5) == 0) ranseeds(:,4:5)=1
    iran0=ranseeds(1,1)
    jran0=ranseeds(1,2)
    kran0=ranseeds(1,3)
    mran0=ranseeds(1,4)
    nran0=ranseeds(1,5)
else if (present(get)) then
    if (lenran == 0) RETURN
    ranseeds(1,1:5)=(/ iran0,jran0,kran0,mran0,nran0 /)
    get=reshape(ranseeds,shape(get))
else if (present(sequence)) then
    call ran_deallocate
    seq=sequence
end if
END SUBROUTINE ran_seed
```

```
ranseeds=reshape(put,shape(ranseeds)) ...
get=reshape(ranseeds,shape(get))
```

Fortran 90's convention is that random state space is a one-dimensional array, so we map to this on both the get and put keywords.

```
iran0=...jran0=...kran0=...
ranseeds(1,1:5)=(/ iran0,jran0,kran0,mran0,nran0 /)
```

It's much more convenient to set a vector from a bunch of scalars then the other way around.

```
SUBROUTINE ran_hash_s(il,ir)
IMPLICIT NONE
INTEGER(K4B), INTENT(INOUT) :: il,ir
```

DES-like hashing of two 32-bit integers, using shifts, xor's, and adds to make the internal nonlinear function.

```
INTEGER(K4B) :: is,j
do j=1,4
    is=ir
    ir=ieor(ir,ishft(ir,5))+1422217823
    ir=ieor(ir,ishft(ir,-16))+1842055030
    ir=ieor(ir,ishft(ir,9))+80567781
    ir=ieor(il,ir)
    il=is
end do
END SUBROUTINE ran_hash_s
```

The various constants are chosen to give good bit mixing and should not be changed.

```
SUBROUTINE ran_hash_v(il,ir)
IMPLICIT NONE
INTEGER(K4B), DIMENSION(:), INTENT(INOUT) :: il,ir
   Vector version of ran_hash_s.
INTEGER(K4B), DIMENSION(size(il)) :: is
INTEGER(K4B) :: j
do j=1,4
    is=ir
    ir=ieor(ir,ishft(ir,5))+1422217823
    ir=ieor(ir,ishft(ir,-16))+1842055030
    ir=ieor(ir,ishft(ir,9))+80567781
    ir=ieor(il,ir)
    il=is
end do
END SUBROUTINE ran_hash_v
```

END MODULE ran_state

The lines

```
ir=ieor(ir,ishft(ir,5))+1422217823
ir=ieor(ir,ishft(ir,-16))+1842055030
ir=ieor(ir,ishft(ir,9))+80567781
```

are *not* a Marsaglia shift sequence, though they resemble one. Instead, they implement a fast, nonlinear function on `ir` that we use as the "S-box" in a DES-like hashing algorithm. (See Volume 1, §7.5.) The triplet $(5, -16, 9)$ is *not* chosen to give a full period Marsaglia sequence — it doesn't. Instead it is chosen as being particularly good at separating in Hamming distance (i.e., number of nonidentical bits) two initially close values of `ir` (e.g., differing by only one bit). The large integer constants are chosen by a similar criterion. Note that the wrap-around of addition without generating an overflow error condition, which was tested in `ran_init`, is relied upon here.

⋆ ⋆ ⋆

```
SUBROUTINE ran0_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,iran0,jran0,kran0,nran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
   Lagged Fibonacci generator combined with a Marsaglia shift sequence. Returns as harvest
   a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This gen-
   erator has the same calling and initialization conventions as Fortran 90's random_number
   routine. Use ran_seed to initialize or reinitialize to a particular sequence. The period of
   this generator is about 2.0 × 10^28, and it fully vectorizes. Validity of the integer model
   assumed by this generator is tested at initialization.
if (lenran < 1) call ran_init(1)                    Initialization routine in ran_state.
rans=iran0-kran0                                    Update Fibonacci generator, which
if (rans < 0) rans=rans+2147483579_k4b                 has period p^2+p+1, p = 2^31 −
iran0=jran0                                            69.
jran0=kran0
kran0=rans
nran0=ieor(nran0,ishft(nran0,13))                   Update Marsaglia shift sequence with
nran0=ieor(nran0,ishft(nran0,-17))                     period 2^32 − 1.
nran0=ieor(nran0,ishft(nran0,5))
rans=ieor(nran0,rans)                               Combine the generators.
harvest=amm*merge(rans,not(rans), rans<0 )          Make the result positive definite (note
END SUBROUTINE ran0_s                                  that amm is negative).
```

```
SUBROUTINE ran0_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,iran,jran,kran,nran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
ranv(1:n)=iran(1:n)-kran(1:n)
where (ranv(1:n) < 0) ranv(1:n)=ranv(1:n)+2147483579_k4b
iran(1:n)=jran(1:n)
jran(1:n)=kran(1:n)
kran(1:n)=ranv(1:n)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
ranv(1:n)=ieor(nran(1:n),ranv(1:n))
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran0_v
```

This is the simplest, and fastest, of the generators provided. It combines a subtractive Fibonacci generator (Number 6 in ref. [1], and one of the generators in Marsaglia and Zaman's `mzran`) with a Marsaglia shift sequence. On typical machines it is only 20% or so faster than `ran1`, however; so we recommend the latter preferentially. While we know of no weakness in `ran0`, we are not offering a prize for finding a weakness. `ran0` does have the feature, useful if you have a machine with nonstandard arithmetic, that it does not go beyond Fortran 90's assumed integer model.

Note that `ran0_s` and `ran0_v` are overloaded by the module `nr` onto the single name `ran0` (and similarly for the routines below).

⋆ ⋆ ⋆

```
SUBROUTINE ran1_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
    iran0,jran0,kran0,nran0,mran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
```

Lagged Fibonacci generator combined with two Marsaglia shift sequences. On output, returns as `harvest` a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90's `random_number` routine. Use `ran_seed` to initialize or reinitialize to a particular sequence. The period of this generator is about 8.5×10^{37}, and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.

```
if (lenran < 1) call ran_init(1)
```
Initialization routine in `ran_state`.

```
rans=iran0-kran0
if (rans < 0) rans=rans+2147483579_k4b
iran0=jran0
jran0=kran0
kran0=rans
```
Update Fibonacci generator, which has period p^2+p+1, $p = 2^{31} - 69$.

```
nran0=ieor(nran0,ishft(nran0,13))
nran0=ieor(nran0,ishft(nran0,-17))
nran0=ieor(nran0,ishft(nran0,5))
```
Update Marsaglia shift sequence.

Once only per cycle, advance sequence by 1, shortening its period to $2^{32} - 2$.

```
if (nran0 == 1) nran0=270369_k4b
mran0=ieor(mran0,ishft(mran0,5))
mran0=ieor(mran0,ishft(mran0,-13))
mran0=ieor(mran0,ishft(mran0,6))
```
Update Marsaglia shift sequence with period $2^{32} - 1$.

```
rans=ieor(nran0,rans)+mran0
```
Combine the generators. The above statement has wrap-around addition.
```
harvest=amm*merge(rans,not(rans), rans<0 )
END SUBROUTINE ran1_s
```
Make the result positive definite (note that amm is negative).

```
SUBROUTINE ran1_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
    iran,jran,kran,nran,mran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
ranv(1:n)=iran(1:n)-kran(1:n)
where (ranv(1:n) < 0) ranv(1:n)=ranv(1:n)+2147483579_k4b
iran(1:n)=jran(1:n)
jran(1:n)=kran(1:n)
kran(1:n)=ranv(1:n)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
where (nran(1:n) == 1) nran(1:n)=270369_k4b
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),5))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),-13))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),6))
ranv(1:n)=ieor(nran(1:n),ranv(1:n))+mran(1:n)
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran1_v
```

The routine ran1 combines *three* fast generators: the two used in ran0, plus an additional (different) Marsaglia shift sequence. The last generator is combined via an addition that can wrap-around.

We think that, within the limits of its floating-point precision, ran1 provides perfect random numbers. We will pay $1000 to the first reader who convinces us otherwise (by exhibiting a statistical test that ran1 fails in a nontrivial way, excluding the ordinary limitations of a floating-point representation).

⋆ ⋆ ⋆

```
SUBROUTINE ran2_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
    iran0,jran0,kran0,nran0,mran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
```
Lagged Fibonacci generator combined with a Marsaglia shift sequence and a linear congruential generator. Returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). This generator has the same calling and initialization conventions as Fortran 90's random_number routine. Use ran_seed to initialize or reinitialize to a particular sequence. The period of this generator is about 8.5×10^{37}, and it fully vectorizes. Validity of the integer model assumed by this generator is tested at initialization.
```
if (lenran < 1) call ran_init(1)
rans=iran0-kran0
if (rans < 0) rans=rans+2147483579_k4b
iran0=jran0
jran0=kran0
kran0=rans
```
Initialization routine in ran_state.

Update Fibonacci generator, which has period p^2+p+1, $p = 2^{31} - 69$.

```
nran0=ieor(nran0,ishft(nran0,13))                    Update Marsaglia shift sequence with
nran0=ieor(nran0,ishft(nran0,-17))                       period 2^32 − 1.
nran0=ieor(nran0,ishft(nran0,5))
rans=iand(mran0,65535)
  Update the sequence m ← 69069m + 820265819 mod 2^32 using shifts instead of multiplies.
  Wrap-around addition (tested at initialization) is used.
mran0=ishft(3533*ishft(mran0,-16)+rans,16)+ &
    3533*rans+820265819_k4b
rans=ieor(nran0,kran0)+mran0                         Combine the generators.
harvest=amm*merge(rans,not(rans), rans<0 )           Make the result positive definite (note
END SUBROUTINE ran2_s                                    that amm is negative).
```

```
SUBROUTINE ran2_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init, &
    iran,jran,kran,nran,mran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
ranv(1:n)=iran(1:n)-kran(1:n)
where (ranv(1:n) < 0) ranv(1:n)=ranv(1:n)+2147483579_k4b
iran(1:n)=jran(1:n)
jran(1:n)=kran(1:n)
kran(1:n)=ranv(1:n)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
ranv(1:n)=iand(mran(1:n),65535)
mran(1:n)=ishft(3533*ishft(mran(1:n),-16)+ranv(1:n),16)+ &
    3533*ranv(1:n)+820265819_k4b
ranv(1:n)=ieor(nran(1:n),kran(1:n))+mran(1:n)
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran2_v
```

ran2, for use by readers whose caution is extreme, also combines three generators. The difference from ran1 is that each generator is based on a completely different method from the other two. The third generator, in this case, is a linear congruential generator, modulo 2^{32}. This generator relies extensively on wrap-around addition (which is automatically tested at initialization). On machines with fast arithmetic, ran2 is on the order of only 20% slower than ran1. We offer a $1000 bounty on ran2, with the same terms as for ran1, above.

⋆ ⋆ ⋆

```
SUBROUTINE expdev_s(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
   Returns in harvest an exponentially distributed, positive, random deviate of unit mean,
   using ran1 as the source of uniform deviates.
REAL(SP) :: dum
call ran1(dum)
harvest=-log(dum)            We use the fact that ran1 never returns exactly 0 or 1.
END SUBROUTINE expdev_s
```

```
SUBROUTINE expdev_v(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
REAL(SP), DIMENSION(size(harvest)) :: dum
call ran1(dum)
harvest=-log(dum)
END SUBROUTINE expdev_v
```

f90 `call ran1(dum)` The only noteworthy thing about this line is its simplicity: Once all the machinery is in place, the random number generators are self-initializing (to the sequence defined by `seq` $= 0$), and (via overloading) usable with both scalar and vector arguments.

$\star \quad \star \quad \star$

```
SUBROUTINE gasdev_s(harvest)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
    Returns in harvest a normally distributed deviate with zero mean and unit variance, using
    ran1 as the source of uniform deviates.
REAL(SP) :: rsq,v1,v2
REAL(SP), SAVE :: g
LOGICAL, SAVE :: gaus_stored=.false.
if (gaus_stored) then                          We have an extra deviate handy,
    harvest=g                                  so return it,
    gaus_stored=.false.                        and unset the flag.
else                                           We don't have an extra deviate handy, so
    do
        call ran1(v1)                          pick two uniform numbers in the square ex-
        call ran1(v2)                              tending from -1 to +1 in each direction,
        v1=2.0_sp*v1-1.0_sp
        v2=2.0_sp*v2-1.0_sp
        rsq=v1**2+v2**2                        see if they are in the unit circle,
        if (rsq > 0.0 .and. rsq < 1.0) exit
    end do                                     otherwise try again.
    rsq=sqrt(-2.0_sp*log(rsq)/rsq)             Now make the Box-Muller transformation to
    harvest=v1*rsq                                 get two normal deviates. Return one and
    g=v2*rsq                                       save the other for next time.
    gaus_stored=.true.                         Set flag.
end if
END SUBROUTINE gasdev_s
```

```
SUBROUTINE gasdev_v(harvest)
USE nrtype; USE nrutil, ONLY : array_copy
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
REAL(SP), DIMENSION(size(harvest)) :: rsq,v1,v2
REAL(SP), ALLOCATABLE, DIMENSION(:), SAVE :: g
INTEGER(I4B) :: n,ng,nn,m
INTEGER(I4B), SAVE :: last_allocated=0
LOGICAL, SAVE :: gaus_stored=.false.
LOGICAL, DIMENSION(size(harvest)) :: mask
n=size(harvest)
if (n /= last_allocated) then
```

```
    if (last_allocated /= 0) deallocate(g)
    allocate(g(n))
    last_allocated=n
    gaus_stored=.false.
end if
if (gaus_stored) then
    harvest=g
    gaus_stored=.false.
else
    ng=1
    do
        if (ng > n) exit
        call ran1(v1(ng:n))
        call ran1(v2(ng:n))
        v1(ng:n)=2.0_sp*v1(ng:n)-1.0_sp
        v2(ng:n)=2.0_sp*v2(ng:n)-1.0_sp
        rsq(ng:n)=v1(ng:n)**2+v2(ng:n)**2
        mask(ng:n)=(rsq(ng:n)>0.0 .and. rsq(ng:n)<1.0)
        call array_copy(pack(v1(ng:n),mask(ng:n)),v1(ng:),nn,m)
        v2(ng:ng+nn-1)=pack(v2(ng:n),mask(ng:n))
        rsq(ng:ng+nn-1)=pack(rsq(ng:n),mask(ng:n))
        ng=ng+nn
    end do
    rsq=sqrt(-2.0_sp*log(rsq)/rsq)
    harvest=v1*rsq
    g=v2*rsq
    gaus_stored=.true.
end if
END SUBROUTINE gasdev_v
```

`if (n /= last_allocated) ...` We make the assumption that, in most cases, the size of `harvest` will not change between successive calls. Therefore, if it *does* change, we don't try to save the previously generated deviates that, half the time, will be around. If your use has rapidly varying sizes (or, even worse, calls alternating between two different sizes), you should remedy this inefficiency in the obvious way.

`call array_copy(pack(v1(ng:n),mask(ng:n)),v1(ng:),nn,m)` This is a variant of the pack-unpack method (see note to `factrl`, p. 1087). Different here is that we don't care which random deviates end up in which component. Thus, we can simply keep packing successful returns into `v1` and `v2` until they are full.

Note also the use of `array_copy`, since we don't know in advance the length of the array returned by `pack`.

⋆ ⋆ ⋆

```
FUNCTION gamdev(ia)
USE nrtype; USE nrutil, ONLY : assert
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ia
REAL(SP) :: gamdev
    Returns a deviate distributed as a gamma distribution of integer order ia, i.e., a waiting
    time to the iath event in a Poisson process of unit mean, using ran1 as the source of
    uniform deviates.
REAL(SP) :: am,e,h,s,x,y,v(2),arr(5)
call assert(ia >= 1, 'gamdev arg')
if (ia < 6) then                                    Use direct method, adding waiting times.
```

```
    call ran1(arr(1:ia))
    x=-log(product(arr(1:ia)))
else                                        Use rejection method.
    do
        call ran1(v)
        v(2)=2.0_sp*v(2)-1.0_sp             These three lines generate the tangent of a
        if (dot_product(v,v) > 1.0) cycle       random angle, i.e., are equivalent to
        y=v(2)/v(1)                             y = tan(πran(idum)).
        am=ia-1
        s=sqrt(2.0_sp*am+1.0_sp)
        x=s*y+am                            We decide whether to reject x:
        if (x <= 0.0) cycle                 Reject in region of zero probability.
        e=(1.0_sp+y**2)*exp(am*log(x/am)-s*y)       Ratio of probability function to
        call ran1(h)                                    comparison function.
        if (h <= e) exit                    Reject on basis of a second uniform deviate.
    end do
end if
gamdev=x
END FUNCTION gamdev
```

`x=-log(product(arr(1:ia)))` Why take the `log` of the product instead of the sum of the `log`s? Because `log` is assumed to be slower than multiply.

We don't have vector versions of the less commonly used deviate generators, `gamdev`, `poidev`, and `bnldev`.

⋆ ⋆ ⋆

```
FUNCTION poidev(xm)
USE nrtype
USE nr, ONLY : gammln,ran1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xm
REAL(SP) :: poidev
    Returns as a floating-point number an integer value that is a random deviate drawn from a
    Poisson distribution of mean xm, using ran1 as a source of uniform random deviates.
REAL(SP) :: em,harvest,t,y
REAL(SP), SAVE :: alxm,g,oldm=-1.0_sp,sq
  oldm is a flag for whether xm has changed since last call.
if (xm < 12.0) then                         Use direct method.
    if (xm /= oldm) then
        oldm=xm
        g=exp(-xm)                          If xm is new, compute the exponential.
    end if
    em=-1
    t=1.0
    do
        em=em+1.0_sp                        Instead of adding exponential deviates it is
        call ran1(harvest)                      equivalent to multiply uniform deviates.
        t=t*harvest                             We never actually have to take the log;
        if (t <= g) exit                        merely compare to the pre-computed ex-
    end do                                      ponential.
else                                        Use rejection method.
    if (xm /= oldm) then                    If xm has changed since the last call, then pre-
        oldm=xm                                 compute some functions that occur be-
        sq=sqrt(2.0_sp*xm)                      low.
        alxm=log(xm)
        g=xm*alxm-gammln(xm+1.0_sp)         The function gammln is the natural log of the
    end if                                      gamma function, as given in §6.1.
    do
```

```
        do
            call ran1(harvest)              y is a deviate from a Lorentzian comparison
            y=tan(PI*harvest)                   function.
            em=sq*y+xm                      em is y, shifted and scaled.
            if (em >= 0.0) exit             Reject if in regime of zero probability.
        end do
        em=int(em)                          The trick for integer-valued distributions.
        t=0.9_sp*(1.0_sp+y**2)*exp(em*alxm-gammln(em+1.0_sp)-g)
          The ratio of the desired distribution to the comparison function; we accept or reject
          by comparing it to another uniform deviate. The factor 0.9 is chosen so that t never
          exceeds 1.
        call ran1(harvest)
        if (harvest <= t) exit
    end do
end if
poidev=em
END FUNCTION poidev
```

⋆ ⋆ ⋆

```
FUNCTION bnldev(pp,n)
USE nrtype
USE nr, ONLY : gammln,ran1
IMPLICIT NONE
REAL(SP), INTENT(IN) :: pp
INTEGER(I4B), INTENT(IN) :: n
REAL(SP) :: bnldev
    Returns as a floating-point number an integer value that is a random deviate drawn from a
    binomial distribution of n trials each of probability pp, using ran1 as a source of uniform
    random deviates.
INTEGER(I4B) :: j
INTEGER(I4B), SAVE :: nold=-1
REAL(SP) :: am,em,g,h,p,sq,t,y,arr(24)
REAL(SP), SAVE :: pc,plog,pclog,en,oldg,pold=-1.0       Arguments from previous calls.
p=merge(pp,1.0_sp-pp, pp <= 0.5_sp )
  The binomial distribution is invariant under changing pp to 1.-pp, if we also change the
  answer to n minus itself; we'll remember to do this below.
am=n*p                                  This is the mean of the deviate to be produced.
if (n < 25) then                        Use the direct method while n is not too large.
    call ran1(arr(1:n))                     This can require up to 25 calls to ran1.
    bnldev=count(arr(1:n)<p)
else if (am < 1.0) then                 If fewer than one event is expected out of 25
    g=exp(-am)                              or more trials, then the distribution is quite
    t=1.0                                   accurately Poisson. Use direct Poisson method.
    do j=0,n
        call ran1(h)
        t=t*h
        if (t < g) exit
    end do
    bnldev=merge(j,n, j <= n)
else                                    Use the rejection method.
    if (n /= nold) then                 If n has changed, then compute useful quanti-
        en=n                                ties.
        oldg=gammln(en+1.0_sp)
        nold=n
    end if
    if (p /= pold) then                 If p has changed, then compute useful quanti-
        pc=1.0_sp-p                         ties.
        plog=log(p)
        pclog=log(pc)
        pold=p
```

```
    end if
    sq=sqrt(2.0_sp*am*pc)                The following code should by now seem familiar:
    do                                       rejection method with a Lorentzian compar-
        call ran1(h)                         ison function.
        y=tan(PI*h)
        em=sq*y+am
        if (em < 0.0 .or. em >= en+1.0_sp) cycle     Reject.
        em=int(em)                       Trick for integer-valued distribution.
        t=1.2_sp*sq*(1.0_sp+y**2)*exp(oldg-gammln(em+1.0_sp)-&
            gammln(en-em+1.0_sp)+em*plog+(en-em)*pclog)
        call ran1(h)
        if (h <= t) exit                 Reject. This happens about 1.5 times per devi-
    end do                                   ate, on average.
    bnldev=em
end if
if (p /= pp) bnldev=n-bnldev             Remember to undo the symmetry transforma-
END FUNCTION bnldev                          tion.
```

⋆ ⋆ ⋆

f90 The routines psdes and psdes_safe both perform *exactly* the same hashing as was done by the Fortran 77 routine psdes. The difference is that psdes makes assumptions about arithmetic that go beyond the strict Fortran 90 model, while psdes_safe makes no such assumptions. The disadvantage of psdes_safe is that it is significantly slower, performing most of its arithmetic in double-precision reals that are then converted to integers with Fortran 90's modulo intrinsic.

In fact the nonsafe version, psdes, works fine on almost all machines and compilers that we have tried. There is a reason for this: Our assumed integer model is the same as the C language unsigned int, and virtually all modern computers and compilers have a lot of C hidden inside. If psdes and psdes_safe produce identical output on your system for any hundred or so different input values, you can be quite confident about using the faster version exclusively.

At the other end of things, note that in the very unlikely case that your system fails on the ran_hash routine in the ran_state module (you will have learned this from error messages generated by ran_init), you can substitute psdes_safe for ran_hash: They are plug-compatible.

```
SUBROUTINE psdes_s(lword,rword)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
    "Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are
    returned hashed on all bits. Note that this version of the routine assumes properties of
    integer arithmetic that go beyond the Fortran 90 model, though they are compatible with
    unsigned integers in C.
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B) :: i,ia,ib,iswap,itmph,itmpl
do i=1,NITER                             Perform niter iterations of DES logic, using a simpler
    iswap=rword                              (noncryptographic) nonlinear function instead of DES's.
    ia=ieor(rword,C1(i))                 The bit-rich constants C1 and (below) C2 guarantee lots
    itmpl=iand(ia,65535)                     of nonlinear mixing.
    itmph=iand(ishft(ia,-16),65535)
```

```
    ib=itmpl**2+not(itmph**2)
    ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
    rword=ieor(lword,ieor(C2(i),ia)+itmpl*itmph)
    lword=iswap
end do
END SUBROUTINE psdes_s



SUBROUTINE psdes_v(lword,rword)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B), DIMENSION(size(lword)) :: ia,ib,iswap,itmph,itmpl
INTEGER(I4B) :: i
i=assert_eq(size(lword),size(rword),'psdes_v')
do i=1,NITER
    iswap=rword
    ia=ieor(rword,C1(i))
    itmpl=iand(ia,65535)
    itmph=iand(ishft(ia,-16),65535)
    ib=itmpl**2+not(itmph**2)
    ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
    rword=ieor(lword,ieor(C2(i),ia)+itmpl*itmph)
    lword=iswap
end do
END SUBROUTINE psdes_v



SUBROUTINE psdes_safe_s(lword,rword)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
```

"Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are returned hashed on all bits. This is a slower version of the routine that makes no assumptions outside of the Fortran 90 integer model.

```
INTEGER(I4B), DIMENSION(4), SAVE :: C1,C2
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B) :: i,ia,ib,iswap
REAL(DP) :: alo,ahi
do i=1,NITER
    iswap=rword
    ia=ieor(rword,C1(i))
    alo=real(iand(ia,65535),dp)
    ahi=real(iand(ishft(ia,-16),65535),dp)
    ib=modint(alo*alo+real(not(modint(ahi*ahi)),dp))
    ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
    rword=ieor(lword,modint(real(ieor(C2(i),ia),dp)+alo*ahi))
    lword=iswap
end do
CONTAINS
FUNCTION modint(x)
REAL(DP), INTENT(IN) :: x
INTEGER(I4B) :: modint
REAL(DP) :: a
REAL(DP), PARAMETER :: big=huge(modint), base=big+big+2.0_dp
a=modulo(x,base)
```

```
if (a > big) a=a-base
modint=nint(a,kind=i4b)
END FUNCTION modint
END SUBROUTINE psdes_safe_s



SUBROUTINE psdes_safe_v(lword,rword)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: lword,rword
INTEGER(I4B), PARAMETER :: NITER=4
INTEGER(I4B), SAVE :: C1(4),C2(4)
DATA C1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',Z'0F33D1B2'/
DATA C2 /Z'4B0F3B58',Z'E874F0C3',Z'6955C5A6',Z'55A7CA46'/
INTEGER(I4B), DIMENSION(size(lword)) :: ia,ib,iswap
REAL(DP), DIMENSION(size(lword)) :: alo,ahi
INTEGER(I4B) :: i
i=assert_eq(size(lword),size(rword),'psdes_safe_v')
do i=1,NITER
    iswap=rword
    ia=ieor(rword,C1(i))
    alo=real(iand(ia,65535),dp)
    ahi=real(iand(ishft(ia,-16),65535),dp)
    ib=modint(alo*alo+real(not(modint(ahi*ahi)),dp))
    ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
    rword=ieor(lword,modint(real(ieor(C2(i),ia),dp)+alo*ahi))
    lword=iswap
end do
CONTAINS

FUNCTION modint(x)
REAL(DP), DIMENSION(:), INTENT(IN) :: x
INTEGER(I4B), DIMENSION(size(x)) :: modint
REAL(DP), DIMENSION(size(x)) :: a
REAL(DP), PARAMETER :: big=huge(modint), base=big+big+2.0_dp
a=modulo(x,base)
where (a > big) a=a-base
modint=nint(a,kind=i4b)
END FUNCTION modint
END SUBROUTINE psdes_safe_v
```

f90 `FUNCTION modint(x)` This embedded routine takes a double-precision real argument, and returns it as an integer mod 2^{32} (correctly wrapping it to negative to take into account that Fortran 90 has no unsigned integers).

⋆ ⋆ ⋆

```
SUBROUTINE ran3_s(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,ran_hash,mran0,nran0,rans
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: harvest
```

Random number generation by DES-like hashing of two 32-bit words, using the algorithm ran_hash. Returns as harvest a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values).

```
INTEGER(K4B) :: temp
if (lenran < 1) call ran_init(1)
nran0=ieor(nran0,ishft(nran0,13))
nran0=ieor(nran0,ishft(nran0,-17))
nran0=ieor(nran0,ishft(nran0,5))
if (nran0 == 1) nran0=270369_k4b
```

Initialize.

Two Marsaglia shift sequences are maintained as input to the hashing. The period of the combined generator is about 1.8×10^{19}.

```
rans=nran0
mran0=ieor(mran0,ishft(mran0,5))
mran0=ieor(mran0,ishft(mran0,-13))
mran0=ieor(mran0,ishft(mran0,6))
temp=mran0
call ran_hash(temp,rans)                          Hash.
harvest=amm*merge(rans,not(rans), rans<0 )        Make the result positive definite (note
END SUBROUTINE ran3_s                                 that amm is negative).
```

```
SUBROUTINE ran3_v(harvest)
USE nrtype
USE ran_state, ONLY: K4B,amm,lenran,ran_init,ran_hash,mran,nran,ranv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
INTEGER(K4B), DIMENSION(size(harvest)) :: temp
INTEGER(K4B) :: n
n=size(harvest)
if (lenran < n+1) call ran_init(n+1)
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),13))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),-17))
nran(1:n)=ieor(nran(1:n),ishft(nran(1:n),5))
where (nran(1:n) == 1) nran(1:n)=270369_k4b
ranv(1:n)=nran(1:n)
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),5))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),-13))
mran(1:n)=ieor(mran(1:n),ishft(mran(1:n),6))
temp=mran(1:n)
call ran_hash(temp,ranv(1:n))
harvest=amm*merge(ranv(1:n),not(ranv(1:n)), ranv(1:n)<0 )
END SUBROUTINE ran3_v
```

As given, `ran3` uses the `ran_hash` function in the module `ran_state` as its DES surrogate. That function is sufficiently fast to make `ran3` only about a factor of 2 slower than our baseline recommended generator `ran1`. The slower routine `psdes` and (even slower) `psdes_safe` are plug-compatible with `ran_hash`, and could be substituted for it in this routine.

⋆ ⋆ ⋆

```
FUNCTION irbit1(iseed)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iseed
INTEGER(I4B) :: irbit1
    Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which
    is modified for the next call).
if (btest(iseed,17) .neqv. btest(iseed,4) .neqv. btest(iseed,1) &
    .neqv. btest(iseed,0)) then
    iseed=ibset(ishft(iseed,1),0)          Leftshift the seed and put a 1 in its bit 1.
    irbit1=1
else                                       But if the XOR calculation gave a 0,
    iseed=ishft(iseed,1)                   then put that in bit 1 instead.
    irbit1=0
end if
END FUNCTION irbit1
```

```
FUNCTION irbit2(iseed)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iseed
INTEGER(I4B) :: irbit2
```

Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which is modified for the next call).

```
INTEGER(I4B), PARAMETER :: IB1=1,IB2=2,IB5=16,MASK=IB1+IB2+IB5
if (btest(iseed,17)) then          Change all masked bits, shift, and put 1 into bit 1.
    iseed=ibset(ishft(ieor(iseed,MASK),1),0)
    irbit2=1
else                               Shift and put 0 into bit 1.
    iseed=ibclr(ishft(iseed,1),0)
    irbit2=0
end if
END FUNCTION irbit2
```

⋆ ⋆ ⋆

```
SUBROUTINE sobseq(x,init)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
INTEGER(I4B), OPTIONAL, INTENT(IN) :: init
INTEGER(I4B), PARAMETER :: MAXBIT=30,MAXDIM=6
```

When the optional integer init is present, internally initializes a set of MAXBIT direction numbers for each of MAXDIM different Sobol' sequences. Otherwise returns as the vector x of length N the next values from N of these sequences. (N must not be changed between initializations.)

```
REAL(SP), SAVE :: fac
INTEGER(I4B) :: i,im,ipp,j,k,l
INTEGER(I4B), DIMENSION(:,:), ALLOCATABLE:: iu
INTEGER(I4B), SAVE :: in
INTEGER(I4B), DIMENSION(MAXDIM), SAVE :: ip,ix,mdeg
INTEGER(I4B), DIMENSION(MAXDIM*MAXBIT), SAVE :: iv
DATA ip /0,1,1,2,1,4/, mdeg /1,2,3,3,4,4/, ix /6*0/
DATA iv /6*1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9,156*0/
if (present(init)) then            Initialize, don't return a vector.
    ix=0
    in=0
    if (iv(1) /= 1) RETURN
    fac=1.0_sp/2.0_sp**MAXBIT
    allocate(iu(MAXDIM,MAXBIT))
    iu=reshape(iv,shape(iu))       To allow both 1D and 2D addressing.
    do k=1,MAXDIM
        do j=1,mdeg(k)             Stored values require only normalization.
            iu(k,j)=iu(k,j)*2**(MAXBIT-j)
        end do
        do j=mdeg(k)+1,MAXBIT      Use the recurrence to get other values.
            ipp=ip(k)
            i=iu(k,j-mdeg(k))
            i=ieor(i,i/2**mdeg(k))
            do l=mdeg(k)-1,1,-1
                if (btest(ipp,0)) i=ieor(i,iu(k,j-l))
                ipp=ipp/2
            end do
            iu(k,j)=i
        end do
    end do
    iv=reshape(iu,shape(iv))
    deallocate(iu)
```

```
else                                        Calculate the next vector in the sequence.
    im=in
    do j=1,MAXBIT                           Find the rightmost zero bit.
        if (.not. btest(im,0)) exit
        im=im/2
    end do
    if (j > MAXBIT) call nrerror('MAXBIT too small in sobseq')
    im=(j-1)*MAXDIM
    j=min(size(x),MAXDIM)
    ix(1:j)=ieor(ix(1:j),iv(1+im:j+im))
      XOR the appropriate direction number into each component of the vector and convert
      to a floating number.
    x(1:j)=ix(1:j)*fac
    in=in+1                                 Increment the counter.
end if
END SUBROUTINE sobseq
```

f90 `if (present(init)) then ... allocate(iu(...)) ... iu=reshape(...)`
Wanting to avoid the deprecated `EQUIVALENCE` statement, we must reshape `iv` into a two-dimensional array, then un-reshape it after we are done. This is done only once, at initialization time, so there is no serious inefficiency introduced.

⋆ ⋆ ⋆

```
SUBROUTINE vegas(region,func,init,ncall,itmx,nprn,tgral,sd,chi2a)
USE nrtype
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: region
INTEGER(I4B), INTENT(IN) :: init,ncall,itmx,nprn
REAL(SP), INTENT(OUT) :: tgral,sd,chi2a
INTERFACE
    FUNCTION func(pt,wgt)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: pt
    REAL(SP), INTENT(IN) :: wgt
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: ALPH=1.5_sp,TINY=1.0e-30_sp
INTEGER(I4B), PARAMETER :: MXDIM=10,NDMX=50
```

Performs Monte Carlo integration of a user-supplied d-dimensional function `func` over a rectangular volume specified by `region`, a vector of length $2d$ consisting of d "lower left" coordinates of the region followed by d "upper right" coordinates. The integration consists of `itmx` iterations, each with approximately `ncall` calls to the function. After each iteration the grid is refined; more than 5 or 10 iterations are rarely useful. The input flag `init` signals whether this call is a new start, or a subsequent call for additional iterations (see comments below). The input flag `nprn` (normally 0) controls the amount of diagnostic output. Returned answers are `tgral` (the best estimate of the integral), `sd` (its standard deviation), and `chi2a` (χ^2 per degree of freedom, an indicator of whether consistent results are being obtained). See text for further details.

```
INTEGER(I4B), SAVE :: i,it,j,k,mds,nd,ndim,ndo,ng,npg      Best make everything static,
INTEGER(I4B), DIMENSION(MXDIM), SAVE :: ia,kg                 allowing restarts.
REAL(SP), SAVE :: calls,dv2g,dxg,f,f2,f2b,fb,rc,ti,tsi,wgt,xjac,xn,xnd,xo,harvest
REAL(SP), DIMENSION(NDMX,MXDIM), SAVE :: d,di,xi
REAL(SP), DIMENSION(MXDIM), SAVE :: dt,dx,x
REAL(SP), DIMENSION(NDMX), SAVE :: r,xin
REAL(DP), SAVE :: schi,si,swgt
```

```
ndim=size(region)/2
if (init <= 0) then                          Normal entry. Enter here on a cold start.
    mds=1                                    Change to mds=0 to disable stratified sam-
    ndo=1                                        pling, i.e., use importance sampling only.
    xi(1,:)=1.0
end if
if (init <= 1) then                          Enter here to inherit the grid from a previous
    si=0.0                                       call, but not its answers.
    swgt=0.0
    schi=0.0
end if
if (init <= 2) then                          Enter here to inherit the previous grid and its
    nd=NDMX                                      answers.
    ng=1
    if (mds /= 0) then                       Set up for stratification.
        ng=(ncall/2.0_sp+0.25_sp)**(1.0_sp/ndim)
        mds=1
        if ((2*ng-NDMX) >= 0) then
            mds=-1
            npg=ng/NDMX+1
            nd=ng/npg
            ng=npg*nd
        end if
    end if
    k=ng**ndim
    npg=max(ncall/k,2)
    calls=real(npg,sp)*real(k,sp)
    dxg=1.0_sp/ng
    dv2g=(calls*dxg**ndim)**2/npg/npg/(npg-1.0_sp)
    xnd=nd
    dxg=dxg*xnd
    dx(1:ndim)=region(1+ndim:2*ndim)-region(1:ndim)
    xjac=1.0_sp/calls*product(dx(1:ndim))
    if (nd /= ndo) then                      Do binning if necessary.
        r(1:max(nd,ndo))=1.0
        do j=1,ndim
            call rebin(ndo/xnd,nd,r,xin,xi(:,j))
        end do
        ndo=nd
    end if
    if (nprn >= 0) write(*,200) ndim,calls,it,itmx,nprn,&
        ALPH,mds,nd,(j,region(j),j,region(j+ndim),j=1,ndim)
end if
do it=1,itmx                                 Main iteration loop. Can enter here (init ≥
    ti=0.0                                       3) to do an additional itmx iterations
    tsi=0.0                                      with all other parameters unchanged.
    kg(:)=1
    d(1:nd,:)=0.0
    di(1:nd,:)=0.0
    iterate: do
        fb=0.0
        f2b=0.0
        do k=1,npg
            wgt=xjac
            do j=1,ndim
                call ran1(harvest)
                xn=(kg(j)-harvest)*dxg+1.0_sp
                ia(j)=max(min(int(xn),NDMX),1)
                if (ia(j) > 1) then
                    xo=xi(ia(j),j)-xi(ia(j)-1,j)
                    rc=xi(ia(j)-1,j)+(xn-ia(j))*xo
                else
                    xo=xi(ia(j),j)
                    rc=(xn-ia(j))*xo
```

```
            end if
            x(j)=region(j)+rc*dx(j)
            wgt=wgt*xo*xnd
        end do
        f=wgt*func(x(1:ndim),wgt)
        f2=f*f
        fb=fb+f
        f2b=f2b+f2
        do j=1,ndim
            di(ia(j),j)=di(ia(j),j)+f
            if (mds >= 0) d(ia(j),j)=d(ia(j),j)+f2
        end do
    end do
    f2b=sqrt(f2b*npg)
    f2b=(f2b-fb)*(f2b+fb)
    if (f2b <= 0.0) f2b=TINY
    ti=ti+fb
    tsi=tsi+f2b
    if (mds < 0) then                    Use stratified sampling.
        do j=1,ndim
            d(ia(j),j)=d(ia(j),j)+f2b
        end do
    end if
    do k=ndim,1,-1
        kg(k)=mod(kg(k),ng)+1
        if (kg(k) /= 1) cycle iterate
    end do
    exit iterate
end do iterate
tsi=tsi*dv2g                             Compute final results for this iteration.
wgt=1.0_sp/tsi
si=si+real(wgt,dp)*real(ti,dp)
schi=schi+real(wgt,dp)*real(ti,dp)**2
swgt=swgt+real(wgt,dp)
tgral=si/swgt
chi2a=max((schi-si*tgral)/(it-0.99_dp),0.0_dp)
sd=sqrt(1.0_sp/swgt)
tsi=sqrt(tsi)
if (nprn >= 0) then
    write(*,201) it,ti,tsi,tgral,sd,chi2a
    if (nprn /= 0) then
        do j=1,ndim
            write(*,202) j,(xi(i,j),di(i,j),&
                i=1+nprn/2,nd,nprn)
        end do
    end if
end if
do j=1,ndim                              Refine the grid. Consult references to under-
    xo=d(1,j)                                stand the subtlety of this procedure. The
    xn=d(2,j)                                refinement is damped, to avoid rapid,
    d(1,j)=(xo+xn)/2.0_sp                    destabilizing changes, and also compressed
    dt(j)=d(1,j)                             in range by the exponent ALPH.
    do i=2,nd-1
        rc=xo+xn
        xo=xn
        xn=d(i+1,j)
        d(i,j)=(rc+xn)/3.0_sp
        dt(j)=dt(j)+d(i,j)
    end do
    d(nd,j)=(xo+xn)/2.0_sp
    dt(j)=dt(j)+d(nd,j)
end do
where (d(1:nd,:) < TINY) d(1:nd,:)=TINY
do j=1,ndim
```

```
            r(1:nd)=((1.0_sp-d(1:nd,j)/dt(j))/(log(dt(j))-log(d(1:nd,j))))**ALPH
            rc=sum(r(1:nd))
            call rebin(rc/xnd,nd,r,xin,xi(:,j))
        end do
    end do
200 format(/' input parameters for vegas: ndim=',i3,' ncall=',f8.0&
        /28x,' it=',i5,' itmx=',i5&
        /28x,' nprn=',i3,' alph=',f5.2/28x,' mds=',i3,' nd=',i4&
        /(30x,'xl(',i2,')= ',g11.4,' xu(',i2,')= ',g11.4))
201 format(/' iteration no.',I3,': ','integral =',g14.7,' +/- ',g9.2,&
        /' all iterations: integral =',g14.7,' +/- ',g9.2,&
        ' chi**2/it''n =',g9.2)
202 format(/' data for axis ',I2/' X delta i ',&
        ' x delta i ',' x delta i ',&
        /(1x,f7.5,1x,g11.4,5x,f7.5,1x,g11.4,5x,f7.5,1x,g11.4))
    CONTAINS
    SUBROUTINE rebin(rc,nd,r,xin,xi)
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: rc
    INTEGER(I4B), INTENT(IN) :: nd
    REAL(SP), DIMENSION(:), INTENT(IN) :: r
    REAL(SP), DIMENSION(:), INTENT(OUT) :: xin
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: xi
```

Utility routine used by vegas, to rebin a vector of densities xi into new bins defined by a vector r.

```
    INTEGER(I4B) :: i,k
    REAL(SP) :: dr,xn,xo
    k=0
    xo=0.0
    dr=0.0
    do i=1,nd-1
        do
            if (rc <= dr) exit
            k=k+1
            dr=dr+r(k)
        end do
        if (k > 1) xo=xi(k-1)
        xn=xi(k)
        dr=dr-rc
        xin(i)=xn-(xn-xo)*dr/r(k)
    end do
    xi(1:nd-1)=xin(1:nd-1)
    xi(nd)=1.0
    END SUBROUTINE rebin
    END SUBROUTINE vegas
```

⋆ ⋆ ⋆

```
    RECURSIVE SUBROUTINE miser(func,regn,ndim,npts,dith,ave,var)
    USE nrtype; USE nrutil, ONLY : assert_eq
    IMPLICIT NONE
    INTERFACE
        FUNCTION func(x)
        USE nrtype
        IMPLICIT NONE
        REAL(SP) :: func
        REAL(SP), DIMENSION(:), INTENT(IN) :: x
        END FUNCTION func
    END INTERFACE
    REAL(SP), DIMENSION(:), INTENT(IN) :: regn
    INTEGER(I4B), INTENT(IN) :: ndim,npts
```

```
REAL(SP), INTENT(IN) :: dith
REAL(SP), INTENT(OUT) :: ave,var
REAL(SP), PARAMETER :: PFAC=0.1_sp,TINY=1.0e-30_sp,BIG=1.0e30_sp
INTEGER(I4B), PARAMETER :: MNPT=15,MNBS=60
```

Monte Carlo samples a user-supplied ndim-dimensional function func in a rectangular volume specified by region, a $2\times$ndim vector consisting of ndim "lower-left" coordinates of the region followed by ndim "upper-right" coordinates. The function is sampled a total of npts times, at locations determined by the method of recursive stratified sampling. The mean value of the function in the region is returned as ave; an estimate of the statistical uncertainty of ave (square of standard deviation) is returned as var. The input parameter dith should normally be set to zero, but can be set to (e.g.) 0.1 if func's active region falls on the boundary of a power-of-2 subdivision of region.

Parameters: PFAC is the fraction of remaining function evaluations used *at each stage* to explore the variance of func. At least MNPT function evaluations are performed in any terminal subregion; a subregion is further bisected only if at least MNBS function evaluations are available.

```
REAL(SP), DIMENSION(:), ALLOCATABLE :: regn_temp
INTEGER(I4B) :: j,jb,n,ndum,npre,nptl,nptr
INTEGER(I4B), SAVE :: iran=0
REAL(SP) :: avel,varl,fracl,fval,rgl,rgm,rgr,&
    s,sigl,siglb,sigr,sigrb,sm,sm2,sumb,sumr
REAL(SP), DIMENSION(:), ALLOCATABLE :: fmaxl,fmaxr,fminl,fminr,pt,rmid
ndum=assert_eq(size(regn),2*ndim,'miser')
allocate(pt(ndim))
if (npts < MNBS) then                          Too few points to bisect; do straight Monte
    sm=0.0                                         Carlo.
    sm2=0.0
    do n=1,npts
        call ranpt(pt,regn)
        fval=func(pt)
        sm=sm+fval
        sm2=sm2+fval**2
    end do
    ave=sm/npts
    var=max(TINY,(sm2-sm**2/npts)/npts**2)
else                                           Do the preliminary (uniform) sampling.
    npre=max(int(npts*PFAC),MNPT)
    allocate(rmid(ndim),fmaxl(ndim),fmaxr(ndim),fminl(ndim),fminr(ndim))
    fminl(:)=BIG                               Initialize the left and right bounds for each
    fminr(:)=BIG                                   dimension.
    fmaxl(:)=-BIG
    fmaxr(:)=-BIG
    do j=1,ndim
        iran=mod(iran*2661+36979,175000)
        s=sign(dith,real(iran-87500,sp))
        rmid(j)=(0.5_sp+s)*regn(j)+(0.5_sp-s)*regn(ndim+j)
    end do
    do n=1,npre                                Loop over the points in the sample.
        call ranpt(pt,regn)
        fval=func(pt)
        where (pt <= rmid)                     Find the left and right bounds for each di-
            fminl=min(fminl,fval)                  mension.
            fmaxl=max(fmaxl,fval)
        elsewhere
            fminr=min(fminr,fval)
            fmaxr=max(fmaxr,fval)
        end where
    end do
    sumb=BIG                                   Choose which dimension jb to bisect.
    jb=0
    siglb=1.0
    sigrb=1.0
    do j=1,ndim
        if (fmaxl(j) > fminl(j) .and. fmaxr(j) > fminr(j)) then
```

```
            sigl=max(TINY,(fmaxl(j)-fminl(j))**(2.0_sp/3.0_sp))
            sigr=max(TINY,(fmaxr(j)-fminr(j))**(2.0_sp/3.0_sp))
            sumr=sigl+sigr                  Equation (7.8.24); see text.
            if (sumr <= sumb) then
                sumb=sumr
                jb=j
                siglb=sigl
                sigrb=sigr
            end if
        end if
    end do
    deallocate(fminr,fminl,fmaxr,fmaxl)
    if (jb == 0) jb=1+(ndim*iran)/175000      MNPT may be too small.
    rgl=regn(jb)                            Apportion the remaining points between left
    rgm=rmid(jb)                                and right.
    rgr=regn(ndim+jb)
    fracl=abs((rgm-rgl)/(rgr-rgl))
    nptl=(MNPT+(npts-npre-2*MNPT)*fracl*siglb/ &      Equation (7.8.23).
        (fracl*siglb+(1.0_sp-fracl)*sigrb))
    nptr=npts-npre-nptl
    allocate(regn_temp(2*ndim))
    regn_temp(:)=regn(:)
    regn_temp(ndim+jb)=rmid(jb)             Set region to left.
    call miser(func,regn_temp,ndim,nptl,dith,avel,varl)
      Dispatch recursive call; will return back here eventually.
    regn_temp(jb)=rmid(jb)
    regn_temp(ndim+jb)=regn(ndim+jb)        Set region to right.
    call miser(func,regn_temp,ndim,nptr,dith,ave,var)
      Dispatch recursive call; will return back here eventually.
    deallocate(regn_temp)
    ave=fracl*avel+(1-fracl)*ave            Combine left and right regions by equation
    var=fracl*fracl*varl+(1-fracl)*(1-fracl)*var      (7.8.11) (1st line).
    deallocate(rmid)
end if
deallocate(pt)
CONTAINS

SUBROUTINE ranpt(pt,region)
USE nr, ONLY : ran1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: pt
REAL(SP), DIMENSION(:), INTENT(IN) :: region
   Returns a uniformly random point pt in a rectangular region of dimension d. Used by
   miser; calls ran1 for uniform deviates.
INTEGER(I4B) :: n
call ran1(pt)
n=size(pt)
pt(1:n)=region(1:n)+(region(n+1:2*n)-region(1:n))*pt(1:n)
END SUBROUTINE ranpt
END SUBROUTINE miser
```

f90 The Fortran 90 version of this routine is much more straightforward than the Fortran 77 version, because Fortran 90 allows recursion. (In fact, this routine is modeled on the C version of `miser`, which was recursive from the start.)

CITED REFERENCES AND FURTHER READING:

Marsaglia, G., and Zaman, A. 1994, *Computers in Physics*, vol. 8, pp. 117–121. [1]

Marsaglia, G. 1985, *Linear Algebra and Its Applications*, vol. 67, pp. 147-156. [2]

Harbison, S.P., and Steele, G.L. 1991, *C: A Reference Manual*, Third Edition, §5.1.1. [3]

Chapter B8. Sorting

Caution! If you are expecting to sort efficiently on a parallel machine, whether its parallelism is small-scale or massive, you almost certainly want to use library routines that are specific to your hardware.

We include in this chapter translations into Fortran 90 of the general purpose *serial* sorting routines that are in Volume 1, augmented by several new routines that give pedagogical demonstrations of how parallel sorts can be achieved with Fortran 90 parallel constructions and intrinsics. However, we intend the above word "pedagogical" to be taken seriously: these new, supposedly parallel, routines are *not* likely to be competitive with machine-specific library routines. Neither do they compete successfully on serial machines with the all-serial routines provided (namely `sort`, `sort2`, `sort3`, `indexx`, and `select`).

⋆ ⋆ ⋆

```
SUBROUTINE sort_pick(arr)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    Sorts an array arr into ascending numerical order, by straight insertion. arr is replaced
    on output by its sorted rearrangement.
INTEGER(I4B) :: i,j,n
REAL(SP) :: a
n=size(arr)
do j=2,n                        Pick out each element in turn.
    a=arr(j)
    do i=j-1,1,-1               Look for the place to insert it.
        if (arr(i) <= a) exit
        arr(i+1)=arr(i)
    end do
    arr(i+1)=a                  Insert it.
end do
END SUBROUTINE sort_pick
```

Not only is `sort_pick` (renamed from Volume 1's `piksrt`) *not parallelizable*, but also, even worse, it is an N^2 routine. It is meant to be invoked only for the most trivial sorting jobs, say, $N < 20$.

⋆ ⋆ ⋆

```
SUBROUTINE sort_shell(arr)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    Sorts an array arr into ascending numerical order by Shell's method (diminishing increment
    sort). arr is replaced on output by its sorted rearrangement.
INTEGER(I4B) :: i,j,inc,n
REAL(SP) :: v
n=size(arr)
inc=1
do                              Determine the starting increment.
    inc=3*inc+1
    if (inc > n) exit
end do
do                              Loop over the partial sorts.
    inc=inc/3
    do i=inc+1,n                Outer loop of straight insertion.
        v=arr(i)
        j=i
        do                      Inner loop of straight insertion.
            if (arr(j-inc) <= v) exit
            arr(j)=arr(j-inc)
            j=j-inc
            if (j <= inc) exit
        end do
        arr(j)=v
    end do
    if (inc <= 1) exit
end do
END SUBROUTINE sort_shell
```

The routine `sort_shell` is renamed from Volume 1's `shell`. Shell's Method, a diminishing increment sort, is not directly parallelizable. However, one can write a fully parallel routine (though not an especially fast one — see remarks at beginning of this chapter) in much the same spirit:

```
SUBROUTINE sort_byreshape(arr)
USE nrtype; USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    Sort an array arr by bubble sorting a succession of reshapings into array slices. The method
    is similar to Shell sort, but allows parallelization within the vectorized masked swap calls.
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: tab
REAL(SP), PARAMETER :: big=huge(arr)
INTEGER(I4B) :: inc,n,m
n=size(arr)
inc=1
do                                      Find the largest increment that fits.
    inc=2*inc+1
    if (inc > n) exit
end do
do                                      Loop over the different shapes for the reshaped
    inc=inc/2                               array.
    m=(n+inc-1)/inc
    allocate(tab(inc,m))                Allocate space and reshape the array. big en-
    tab=reshape(arr, (/inc,m/) , (/big/) )      sures that fill elements stay at the
    do                                          end.
          Bubble sort all the rows in parallel.
        call swap(tab(:,1:m-1:2),tab(:,2:m:2), &
            tab(:,1:m-1:2)>tab(:,2:m:2))
        call swap(tab(:,2:m-1:2),tab(:,3:m:2), &
            tab(:,2:m-1:2)>tab(:,3:m:2))
```

```
        if (all(tab(:,1:m-1) <= tab(:,2:m))) exit
    end do
    arr=reshape(tab,shape(arr))          Put the array back together for the next shape.
    deallocate(tab)
    if (inc <= 1) exit
end do
END SUBROUTINE sort_byreshape
```

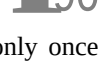
The basic idea is to reshape the given one-dimensional array into a succession of two-dimensional arrays, starting with "tall and narrow" (many rows, few columns), and ending up with "short and wide" (many columns, few rows). At each stage we sort all the rows in parallel by a bubble sort, giving something close to Shell's diminishing increments.

⋆ ⋆ ⋆

We now arrive at those routines, based on the Quicksort algorithm, that we actually intend for use with general N on serial machines:

```
SUBROUTINE sort(arr)
USE nrtype; USE nrutil, ONLY : swap,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
    Sorts an array arr into ascending numerical order using the Quicksort algorithm. arr is
    replaced on output by its sorted rearrangement.
    Parameters: NN is the size of subarrays sorted by straight insertion and NSTACK is the
    required auxiliary storage.
REAL(SP) :: a
INTEGER(I4B) :: n,k,i,j,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=size(arr)
jstack=0
l=1
r=n
do
    if (r-l < NN) then                   Insertion sort when subarray small enough.
        do j=l+1,r
            a=arr(j)
            do i=j-1,l,-1
                if (arr(i) <= a) exit
                arr(i+1)=arr(i)
            end do
            arr(i+1)=a
        end do
        if (jstack == 0) RETURN
        r=istack(jstack)                 Pop stack and begin a new round of partition-
        l=istack(jstack-1)                   ing.
        jstack=jstack-2
    else                                 Choose median of left, center, and right elements
        k=(l+r)/2                            as partitioning element a. Also rearrange so
        call swap(arr(k),arr(l+1))           that a(l) ≤ a(l+1) ≤ a(r).
        call swap(arr(l),arr(r),arr(l)>arr(r))
        call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
        call swap(arr(l),arr(l+1),arr(l)>arr(l+1))
        i=l+1                            Initialize pointers for partitioning.
        j=r
        a=arr(l+1)                       Partitioning element.
        do                               Here is the meat.
            do                           Scan up to find element >= a.
                i=i+1
```

```
            if (arr(i) >= a) exit
        end do
        do                              Scan down to find element <= a.
            j=j-1
            if (arr(j) <= a) exit
        end do
        if (j < i) exit                 Pointers crossed. Exit with partitioning complete.
        call swap(arr(i),arr(j))            Exchange elements.
    end do
    arr(l+1)=arr(j)                     Insert partitioning element.
    arr(j)=a
    jstack=jstack+2
      Push pointers to larger subarray on stack; process smaller subarray immediately.
    if (jstack > NSTACK) call nrerror('sort: NSTACK too small')
    if (r-i+1 >= j-l) then
        istack(jstack)=r
        istack(jstack-1)=i
        r=j-1
    else
        istack(jstack)=j-1
        istack(jstack-1)=l
        l=i
    end if
  end if
end do
END SUBROUTINE sort
```



`call swap(...) ... call swap(...)` One might think twice about putting all these external function calls (to `nrutil` routines) in the inner loop of something as streamlined as a sort routine, but here they are executed only once for each partitioning.

`call swap(arr(i),arr(j))` This call *is* in a loop, but not the innermost loop. Most modern machines are very fast at the "context changes" implied by subroutine calls and returns; but in a time-critical context you might code this swap in-line and see if there is any timing difference.

```
SUBROUTINE sort2(arr,slave)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave
    Sorts an array arr into ascending order using Quicksort, while making the corresponding
    rearrangement of the same-size array slave. The sorting and rearrangement are performed
    by means of an index array.
INTEGER(I4B) :: ndum
INTEGER(I4B), DIMENSION(size(arr)) :: index
ndum=assert_eq(size(arr),size(slave),'sort2')
call indexx(arr,index)          Make the index array.
arr=arr(index)                  Sort arr.
slave=slave(index)              Rearrange slave.
END SUBROUTINE sort2
```

⋆ ⋆ ⋆

A close surrogate for the Quicksort partition-exchange algorithm can be coded, parallelizable, by using Fortran 90's `pack` intrinsic. On real compilers, unfortunately, the resulting code is not very efficient as compared with (on serial machines) the tightness of `sort`'s inner loop, above, or (on parallel machines) supplied library sort routines. We illustrate the principle nevertheless in the following routine.

```
RECURSIVE SUBROUTINE sort_bypack(arr)
USE nrtype; USE nrutil, ONLY : array_copy,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
```
Sort an array arr by recursively applying the Fortran 90 pack intrinsic. The method is similar to Quicksort, but this variant allows parallelization by the Fortran 90 compiler.
```
REAL(SP) :: a
INTEGER(I4B) :: n,k,nl,nerr
INTEGER(I4B), SAVE :: level=0
LOGICAL, DIMENSION(:), ALLOCATABLE, SAVE :: mask
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: temp
n=size(arr)
if (n <= 1) RETURN
k=(1+n)/2
call swap(arr(1),arr(k),arr(1)>arr(k))          Pivot element is median of first, middle,
call swap(arr(k),arr(n),arr(k)>arr(n))              and last.
call swap(arr(1),arr(k),arr(1)>arr(k))
if (n <= 3) RETURN
level=level+1                                   Keep track of recursion level to avoid al-
if (level == 1) allocate(mask(n),temp(n))           location overhead.
a=arr(k)
mask(1:n) = (arr <= a)                          Which elements move to left?
mask(k) = .false.
call array_copy(pack(arr,mask(1:n)),temp,nl,nerr)       Move them.
mask(k) = .true.
temp(nl+2:n)=pack(arr,.not. mask(1:n))          Move others to right.
temp(nl+1)=a
arr=temp(1:n)
call sort_bypack(arr(1:nl))                     And recurse.
call sort_bypack(arr(nl+2:n))
if (level == 1) deallocate(mask,temp)
level=level-1
END SUBROUTINE sort_bypack
```

⋆ ⋆ ⋆

The following routine, sort_heap, is renamed from Volume 1's hpsort.

```
SUBROUTINE sort_heap(arr)
USE nrtype
USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
```
Sorts an array arr into ascending numerical order using the Heapsort algorithm. arr is replaced on output by its sorted rearrangement.
```
INTEGER(I4B) :: i,n
n=size(arr)
do i=n/2,1,-1
```
The index i, which here determines the "left" range of the sift-down, i.e., the element to be sifted down, is decremented from n/2 down to 1 during the "hiring" (heap creation) phase.
```
    call sift_down(i,n)
end do
do i=n,2,-1
```
Here the "right" range of the sift-down is decremented from n-1 down to 1 during the "retirement-and-promotion" (heap selection) phase.
```
    call swap(arr(1),arr(i))                    Clear a space at the end of the array, and
    call sift_down(1,i-1)                           retire the top of the heap into it.
end do
CONTAINS

SUBROUTINE sift_down(l,r)
INTEGER(I4B), INTENT(IN) :: l,r
```

```
    Carry out the sift-down on element arr(l) to maintain the heap structure.
INTEGER(I4B) :: j,jold
REAL(SP) :: a
a=arr(l)
jold=l
j=l+l
do                                          "Do while j <= r:"
    if (j > r) exit
    if (j < r) then
        if (arr(j) < arr(j+1)) j=j+1        Compare to the better underling.
    end if
    if (a >= arr(j)) exit                   Found a's level. Terminate the sift-down. Oth-
    arr(jold)=arr(j)                            erwise, demote a and continue.
    jold=j
    j=j+j
end do
arr(jold)=a                                 Put a into its slot.
END SUBROUTINE sift_down
END SUBROUTINE sort_heap
```

⋆ ⋆ ⋆

Another opportunity provided by Fortran 90 for a fully parallelizable sort, at least pedagogically, is to use the language's allowed access to the actual floating-point representation and to code a radix sort [1] on its bits. This is *not* efficient, but it illustrates some Fortran 90 language features perhaps worthy of study for other applications.

```
SUBROUTINE sort_radix(arr)
USE nrtype; USE nrutil, ONLY : array_copy,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
   Sort an array arr by radix sort on its bits.
INTEGER(I4B), DIMENSION(size(arr)) :: narr,temp
LOGICAL, DIMENSION(size(arr)) :: msk
INTEGER(I4B) :: k,negm,ib,ia,n,nl,nerr
  Because we are going to transfer reals to integers, we must check that the number of bits
  is the same in each:
ib=bit_size(narr)
ia=ceiling(log(real(maxexponent(arr)-minexponent(arr),sp))/log(2.0_sp)) &
    + digits(arr)
if (ib /= ia) call nrerror('sort_radix: bit sizes not compatible')
negm=not(ishftc(1,-1))                      Mask for all bits except sign bit.
n=size(arr)
narr=transfer(arr,narr,n)
where (btest(narr,ib-1)) narr=ieor(narr,negm)   Flip all bits on neg. numbers.
do k=0,ib-2
      Work from low- to high-order bits, and partition the array according to the value of the
      bit.
    msk=btest(narr,k)
    call array_copy(pack(narr,.not. msk),temp,nl,nerr)
    temp(nl+1:n)=pack(narr,msk)
    narr=temp
end do
msk=btest(narr,ib-1)                        The sign bit gets separate treat-
call array_copy(pack(narr,msk),temp,nl,nerr)    ment, since here 1 comes be-
temp(nl+1:n)=pack(narr,.not. msk)               fore 0.
narr=temp
where (btest(narr,ib-1)) narr=ieor(narr,negm)   Unflip all bits on neg. numbers.
arr=transfer(narr,arr,n)
END SUBROUTINE sort_radix
```

⋆ ⋆ ⋆

We overload the generic name indexx with two specific implementations, one for SP floating values, the other for I4B integers. (You can of course add more overloadings if you need them.)

```
SUBROUTINE indexx_sp(arr,index)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
```

Indexes an array arr, i.e., outputs the array index of length N such that arr(index(j)) is in ascending order for $j = 1, 2, \ldots, N$. The input quantity arr is not changed.

```
REAL(SP) :: a
INTEGER(I4B) :: n,k,i,j,indext,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=assert_eq(size(index),size(arr),'indexx_sp')
index=arth(1,1,n)
jstack=0
l=1
r=n
do
    if (r-l < NN) then
        do j=l+1,r
            indext=index(j)
            a=arr(indext)
            do i=j-1,l,-1
                if (arr(index(i)) <= a) exit
                index(i+1)=index(i)
            end do
            index(i+1)=indext
        end do
        if (jstack == 0) RETURN
        r=istack(jstack)
        l=istack(jstack-1)
        jstack=jstack-2
    else
        k=(l+r)/2
        call swap(index(k),index(l+1))
        call icomp_xchg(index(l),index(r))
        call icomp_xchg(index(l+1),index(r))
        call icomp_xchg(index(l),index(l+1))
        i=l+1
        j=r
        indext=index(l+1)
        a=arr(indext)
        do
            do
                i=i+1
                if (arr(index(i)) >= a) exit
            end do
            do
                j=j-1
                if (arr(index(j)) <= a) exit
            end do
            if (j < i) exit
            call swap(index(i),index(j))
        end do
        index(l+1)=index(j)
        index(j)=indext
        jstack=jstack+2
        if (jstack > NSTACK) call nrerror('indexx: NSTACK too small')
        if (r-i+1 >= j-l) then
            istack(jstack)=r
```

```
            istack(jstack-1)=i
            r=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        end if
    end if
end do
CONTAINS
SUBROUTINE icomp_xchg(i,j)
INTEGER(I4B), INTENT(INOUT) :: i,j
INTEGER(I4B) :: swp
if (arr(j) < arr(i)) then
    swp=i
    i=j
    j=swp
end if
END SUBROUTINE icomp_xchg
END SUBROUTINE indexx_sp



SUBROUTINE indexx_i4b(iarr,index)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror,swap
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
INTEGER(I4B) :: a
INTEGER(I4B) :: n,k,i,j,indext,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=assert_eq(size(index),size(iarr),'indexx_sp')
index=arth(1,1,n)
jstack=0
l=1
r=n
do
    if (r-l < NN) then
        do j=l+1,r
            indext=index(j)
            a=iarr(indext)
            do i=j-1,1,-1
                if (iarr(index(i)) <= a) exit
                index(i+1)=index(i)
            end do
            index(i+1)=indext
        end do
        if (jstack == 0) RETURN
        r=istack(jstack)
        l=istack(jstack-1)
        jstack=jstack-2
    else
        k=(l+r)/2
        call swap(index(k),index(l+1))
        call icomp_xchg(index(l),index(r))
        call icomp_xchg(index(l+1),index(r))
        call icomp_xchg(index(l),index(l+1))
        i=l+1
        j=r
        indext=index(l+1)
        a=iarr(indext)
        do
            do
```

```
            i=i+1
            if (iarr(index(i)) >= a) exit
        end do
        do
            j=j-1
            if (iarr(index(j)) <= a) exit
        end do
        if (j < i) exit
        call swap(index(i),index(j))
    end do
    index(l+1)=index(j)
    index(j)=indext
    jstack=jstack+2
    if (jstack > NSTACK) call nrerror('indexx: NSTACK too small')
    if (r-i+1 >= j-l) then
        istack(jstack)=r
        istack(jstack-1)=i
        r=j-1
    else
        istack(jstack)=j-1
        istack(jstack-1)=l
        l=i
    end if
  end if
end do
CONTAINS
SUBROUTINE icomp_xchg(i,j)
INTEGER(I4B), INTENT(INOUT) :: i,j
INTEGER(I4B) :: swp
if (iarr(j) < iarr(i)) then
  swp=i
  i=j
  j=swp
end if
END SUBROUTINE icomp_xchg
END SUBROUTINE indexx_i4b
```

★ ★ ★

```
SUBROUTINE sort3(arr,slave1,slave2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave1,slave2
```

Sorts an array arr into ascending order using Quicksort, while making the corresponding rearrangement of the same-size arrays slave1 and slave2. The sorting and rearrangement are performed by means of an index array.

```
INTEGER(I4B) :: ndum
INTEGER(I4B), DIMENSION(size(arr)) :: index
ndum=assert_eq(size(arr),size(slave1),size(slave2),'sort3')
call indexx(arr,index)          Make the index array.
arr=arr(index)                  Sort arr.
slave1=slave1(index)            Rearrange slave1,
slave2=slave2(index)            and slave2.
END SUBROUTINE sort3
```

★ ★ ★

```
FUNCTION rank(index)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: index
INTEGER(I4B), DIMENSION(size(index)) :: rank
```

Given index as output from the routine indexx, this routine returns a same-size array rank, the corresponding table of ranks.

```
rank(index(:))=arth(1,1,size(index))
END FUNCTION rank
```

⋆ ⋆ ⋆

Just as in the case of sort, where an approximation of the underlying Quicksort partition-exchange algorithm can be captured with the Fortran 90 pack intrinsic, the same can be done with indexx. As before, although it is in principle parallelizable by the compiler, it is likely not competitive with library routines.

```
RECURSIVE SUBROUTINE index_bypack(arr,index,partial)
USE nrtype; USE nrutil, ONLY : array_copy,arth,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: index
INTEGER, OPTIONAL, INTENT(IN) :: partial
```

Indexes an array arr, i.e., outputs the array index of length N such that arr(index(j)) is in ascending order for $j = 1, 2, \ldots, N$. The method is to apply recursively the Fortran 90 pack intrinsic. This is similar to Quicksort, but allows parallelization by the Fortran 90 compiler. partial is an optional argument that is used only internally on the recursive calls.

```
REAL(SP) :: a
INTEGER(I4B) :: n,k,nl,indext,nerr
INTEGER(I4B), SAVE :: level=0
LOGICAL, DIMENSION(:), ALLOCATABLE, SAVE :: mask
INTEGER(I4B), DIMENSION(:), ALLOCATABLE, SAVE :: temp
if (present(partial)) then
    n=size(index)
else
    n=assert_eq(size(index),size(arr),'indexx_bypack')
    index=arth(1,1,n)
end if
if (n <= 1) RETURN
k=(1+n)/2
call icomp_xchg(index(1),index(k))                  Pivot element is median of first, mid-
call icomp_xchg(index(k),index(n))                      dle, and last.
call icomp_xchg(index(1),index(k))
if (n <= 3) RETURN
level=level+1                                       Keep track of recursion level to avoid
if (level == 1) allocate(mask(n),temp(n))               allocation overhead.
indext=index(k)
a=arr(indext)
mask(1:n) = (arr(index) <= a)                       Which elements move to left?
mask(k) = .false.
call array_copy(pack(index,mask(1:n)),temp,nl,nerr)     Move them.
mask(k) = .true.
temp(nl+2:n)=pack(index,.not. mask(1:n))            Move others to right.
temp(nl+1)=indext
index=temp(1:n)
call index_bypack(arr,index(1:nl),partial=1)        And recurse.
call index_bypack(arr,index(nl+2:n),partial=1)
if (level == 1) deallocate(mask,temp)
level=level-1
```

```
CONTAINS
SUBROUTINE icomp_xchg(i,j)
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: i,j
```
Swap or don't swap integer arguments, depending on the ordering of their corresponding elements in an array arr.
```
INTEGER(I4B) :: swp
if (arr(j) < arr(i)) then
    swp=i
    i=j
    j=swp
end if
END SUBROUTINE icomp_xchg
END SUBROUTINE index_bypack
```

⋆ ⋆ ⋆

```
FUNCTION select(k,arr)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
REAL(SP) :: select
```
Returns the kth smallest value in the array arr. The input array will be rearranged to have this value in location arr(k), with all smaller elements moved to arr(1:k-1) (in arbitrary order) and all larger elements in arr(k+1:) (also in arbitrary order).
```
INTEGER(I4B) :: i,r,j,l,n
REAL(SP) :: a
n=size(arr)
call assert(k >= 1, k <= n, 'select args')
l=1
r=n
do
    if (r-l <= 1) then                  Active partition contains 1 or 2 elements.
        if (r-l == 1) call swap(arr(l),arr(r),arr(l)>arr(r))   Active partition contains 2 elements.
        select=arr(k)
        RETURN
    else                                Choose median of left, center, and right elements
        i=(l+r)/2                           as partitioning element a. Also rearrange so
        call swap(arr(i),arr(l+1))          that arr(l) ≤ arr(l+1) ≤ arr(r).
        call swap(arr(l),arr(r),arr(l)>arr(r))
        call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
        call swap(arr(l),arr(l+1),arr(l)>arr(l+1))
        i=l+1                           Initialize pointers for partitioning.
        j=r
        a=arr(l+1)                      Partitioning element.
        do                              Here is the meat.
            do                          Scan up to find element > a.
                i=i+1
                if (arr(i) >= a) exit
            end do
            do                          Scan down to find element < a.
                j=j-1
                if (arr(j) <= a) exit
            end do
            if (j < i) exit             Pointers crossed. Exit with partitioning complete.
            call swap(arr(i),arr(j))        Exchange elements.
        end do
        arr(l+1)=arr(j)                 Insert partitioning element.
        arr(j)=a
        if (j >= k) r=j-1               Keep active the partition that contains the kth
                                            element.
```

```
        if (j <= k) l=i
    end if
end do
END FUNCTION select
```

⋆ ⋆ ⋆

The following routine, `select_inplace`, is renamed from Volume 1's `selip`.

```
FUNCTION select_inplace(k,arr)
USE nrtype
USE nr, ONLY : select
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP) :: select_inplace
    Returns the kth smallest value in the array arr, without altering the input array. In Fortran
    90's assumed memory-rich environment, we just call select in scratch space.
REAL(SP), DIMENSION(size(arr)) :: tarr
tarr=arr
select_inplace=select(k,tarr)
END FUNCTION select_inplace
```

f90 Volume 1's `selip` routine uses an entirely different algorithm, for the purpose of avoiding any additional memory allocation beyond that of the input array. Fortran 90 presumes a richer memory environment, so `select_inplace` simply does the obvious (destructive) selection in scratch space. You can of course use the old `selip` if your in-core or in-cache memory is at a premium.

```
FUNCTION select_bypack(k,arr)
USE nrtype; USE nrutil, ONLY : array_copy,assert,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
REAL(SP) :: select_bypack
    Returns the kth smallest value in the array arr. The input array will be rearranged to have
    this value in location arr(k), with all smaller elements moved to arr(1:k-1) (in arbitrary
    order) and all larger elements in arr(k+1:) (also in arbitrary order). This implementation
    allows parallelization in the Fortran 90 pack intrinsic.
LOGICAL, DIMENSION(size(arr)) :: mask
REAL(SP), DIMENSION(size(arr)) :: temp
INTEGER(I4B) :: i,r,j,l,n,nl,nerr
REAL(SP) :: a
n=size(arr)
call assert(k >= 1, k <= n, 'select_bypack args')
l=1                                                    Initial left and right bounds.
r=n
do                                                     Keep partitioning until desired el-
    if (r-l <= 1) exit                                     ement is found.
    i=(l+r)/2
    call swap(arr(l),arr(i),arr(l)>arr(i))             Pivot element is median of first,
    call swap(arr(i),arr(r),arr(i)>arr(r))                 middle, and last.
    call swap(arr(l),arr(i),arr(l)>arr(i))
    a=arr(i)
    mask(l:r) = (arr(l:r) <= a)                        Which elements move to left?
    mask(i) = .false.
    call array_copy(pack(arr(l:r),mask(l:r)),temp(l:),nl,nerr)     Move them.
    j=l+nl
```

```
    mask(i) = .true.
    temp(j+1:r)=pack(arr(l:r),.not. mask(l:r))      Move others to right.
    temp(j)=a
    arr(l:r)=temp(l:r)
    if (k > j) then                                 Reset bounds to whichever side
        l=j+1                                           has the desired element.
    else if (k < j) then
        r=j-1
    else
        l=j
        r=j
    end if
end do
if (r-l == 1) call swap(arr(l),arr(r),arr(l)>arr(r))    Case of only two left.
select_bypack=arr(k)
END FUNCTION select_bypack
```

The above routine `select_bypack` is parallelizable, but as discussed above (`sort_bypack`, `index_bypack`) it is generally not very efficient.

⋆ ⋆ ⋆

The following routine, `select_heap`, is renamed from Volume 1's `hpsel`.

```
SUBROUTINE select_heap(arr,heap)
USE nrtype; USE nrutil, ONLY : nrerror,swap
USE nr, ONLY : sort
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP), DIMENSION(:), INTENT(OUT) :: heap
    Returns in heap, an array of length M, the largest M elements of the array arr of length
    N, with heap(1) guaranteed to be the the Mth largest element. The array arr is not
    altered. For efficiency, this routine should be used only when M ≪ N.
INTEGER(I4B) :: i,j,k,m,n
m=size(heap)
n=size(arr)
if (m > n/2 .or. m < 1) call nrerror('probable misuse of select_heap')
heap=arr(1:m)
call sort(heap)                         Create initial heap by overkill! We assume m ≪ n.
do i=m+1,n                              For each remaining element...
    if (arr(i) > heap(1)) then              Put it on the heap?
        heap(1)=arr(i)
        j=1
        do                              Sift down.
            k=2*j
            if (k > m) exit
            if (k /= m) then
                if (heap(k) > heap(k+1)) k=k+1
            end if
            if (heap(j) <= heap(k)) exit
            call swap(heap(k),heap(j))
            j=k
        end do
    end if
end do
END SUBROUTINE select_heap
```

⋆ ⋆ ⋆

```
FUNCTION eclass(lista,listb,n)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: lista,listb
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclass
```

Given M equivalences between pairs of `n` individual elements in the form of the input arrays `lista` and `listb` of length M, this routine returns in an array of length `n` the number of the equivalence class of each of the `n` elements, integers between 1 and `n` (not all such integers used).

```
INTEGER :: j,k,l,m
m=assert_eq(size(lista),size(listb),'eclass')
eclass(1:n)=arth(1,1,n)              Initialize each element its own class.
do l=1,m                             For each piece of input information...
    j=lista(l)
    do                               Track first element up to its ancestor.
        if (eclass(j) == j) exit
        j=eclass(j)
    end do
    k=listb(l)
    do                               Track second element up to its ancestor.
        if (eclass(k) == k) exit
        k=eclass(k)
    end do
    if (j /= k) eclass(j)=k          If they are not already related, make them so.
end do
do j=1,n                             Final sweep up to highest ancestors.
    do
        if (eclass(j) == eclass(eclass(j))) exit
        eclass(j)=eclass(eclass(j))
    end do
end do
END FUNCTION eclass

FUNCTION eclazz(equiv,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTERFACE
    FUNCTION equiv(i,j)
    USE nrtype
    IMPLICIT NONE
    LOGICAL(LGT) :: equiv
    INTEGER(I4B), INTENT(IN) :: i,j
    END FUNCTION equiv
END INTERFACE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclazz
```

Given a user-supplied logical function `equiv` that tells whether a pair of elements, each in the range `1...n`, are related, return in an array of length `n` equivalence class numbers for each element.

```
INTEGER :: i,j
eclazz(1:n)=arth(1,1,n)
do i=2,n                                  Loop over first element of all pairs.
    do j=1,i-1                            Loop over second element of all pairs.
        eclazz(j)=eclazz(eclazz(j))       Sweep it up this much.
        if (equiv(i,j)) eclazz(eclazz(eclazz(j)))=i
          Good exercise for the reader to figure out why this much ancestry is necessary!
    end do
end do
do i=1,n                                  Only this much sweeping is needed finally.
    eclazz(i)=eclazz(eclazz(i))
end do
END FUNCTION eclazz
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.5. [1]

Chapter B9. Root Finding and Nonlinear Sets of Equations

```
SUBROUTINE scrsho(func)
USE nrtype
IMPLICIT NONE
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ISCR=60,JSCR=21
```

For interactive "dumb terminal" use. Produce a crude graph of the function `func` over the prompted-for interval `x1,x2`. Query for another plot until the user signals satisfaction. Parameters: Number of horizontal and vertical positions in display.

```
INTEGER(I4B) :: i,j,jz
REAL(SP) :: dx,dyj,x,x1,x2,ybig,ysml
REAL(SP), DIMENSION(ISCR) :: y
CHARACTER(1), DIMENSION(ISCR,JSCR) :: scr
CHARACTER(1) :: blank=' ',zero='-',yy='l',xx='-',ff='x'
do
    write (*,*) ' Enter x1,x2 (= to stop)'        Query for another plot; quit if x1=x2.
    read (*,*) x1,x2
    if (x1 == x2) RETURN
    scr(1,1:JSCR)=yy                              Fill vertical sides with character 'l'.
    scr(ISCR,1:JSCR)=yy
    scr(2:ISCR-1,1)=xx                            Fill top, bottom with character '-'.
    scr(2:ISCR-1,JSCR)=xx
    scr(2:ISCR-1,2:JSCR-1)=blank                  Fill interior with blanks.
    dx=(x2-x1)/(ISCR-1)
    x=x1
    do i=1,ISCR                                   Evaluate the function at equal intervals.
        y(i)=func(x)
        x=x+dx
    end do
    ysml=min(minval(y(:)),0.0_sp)                 Limits will include 0.
    ybig=max(maxval(y(:)),0.0_sp)
    if (ybig == ysml) ybig=ysml+1.0               Be sure to separate top and bottom.
    dyj=(JSCR-1)/(ybig-ysml)
    jz=1-ysml*dyj                                 Note which row corresponds to 0.
    scr(1:ISCR,jz)=zero
    do i=1,ISCR                                   Place an indicator at function height and 0.
        j=1+(y(i)-ysml)*dyj
        scr(i,j)=ff
    end do
    write (*,'(1x,1p,e10.3,1x,80a1)') ybig,(scr(i,JSCR),i=1,ISCR)
    do j=JSCR-1,2,-1                              Display.
        write (*,'(12x,80a1)') (scr(i,j),i=1,ISCR)
    end do
    write (*,'(1x,1p,e10.3,1x,80a1)') ysml,(scr(i,1),i=1,ISCR)
```

```
    write (*,'(12x,1p,e10.3,40x,e10.3)') x1,x2
end do
END SUBROUTINE scrsho
```

f90 `CHARACTER(1), DIMENSION(ISCR,JSCR) :: scr` In Fortran 90, the length of variables of type character should be declared as `CHARACTER(1)` or `CHARACTER(len=1)` (for a variable of length 1), rather than the older form `CHARACTER*1`. While the older form is still legal syntax, the newer one is more consistent with the syntax of other type declarations. (For variables of length 1, you can actually omit the length specifier entirely, and just say `CHARACTER`.)

⋆ ⋆ ⋆

```
SUBROUTINE zbrac(func,x1,x2,succes)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: x1,x2
LOGICAL(LGT), INTENT(OUT) :: succes
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NTRY=50
REAL(SP), PARAMETER :: FACTOR=1.6_sp
```

Given a function `func` and an initial guessed range `x1` to `x2`, the routine expands the range geometrically until a root is bracketed by the returned values `x1` and `x2` (in which case `succes` returns as `.true.`) or until the range becomes unacceptably large (in which case `succes` returns as `.false.`).

```
INTEGER(I4B) :: j
REAL(SP) :: f1,f2
if (x1 == x2) call nrerror('zbrac: you have to guess an initial range')
f1=func(x1)
f2=func(x2)
succes=.true.
do j=1,NTRY
    if ((f1 > 0.0 .and. f2 < 0.0) .or. &
        (f1 < 0.0 .and. f2 > 0.0)) RETURN
    if (abs(f1) < abs(f2)) then
        x1=x1+FACTOR*(x1-x2)
        f1=func(x1)
    else
        x2=x2+FACTOR*(x2-x1)
        f2=func(x2)
    end if
end do
succes=.false.
END SUBROUTINE zbrac
```

⋆ ⋆ ⋆

```
SUBROUTINE zbrak(func,x1,x2,n,xb1,xb2,nb)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), INTENT(OUT) :: nb
REAL(SP), INTENT(IN) :: x1,x2
REAL(SP), DIMENSION(:), POINTER :: xb1,xb2
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
```

Given a function `func` defined on the interval from `x1`–`x2` subdivide the interval into `n` equally spaced segments, and search for zero crossings of the function. `nb` is returned as the number of bracketing pairs `xb1(1:nb)`, `xb2(1:nb)` that are found. `xb1` and `xb2` are pointers to arrays of length `nb` that are dynamically allocated by the routine.

```
INTEGER(I4B) :: i
REAL(SP) :: dx
REAL(SP), DIMENSION(0:n) :: f,x
LOGICAL(LGT), DIMENSION(1:n) :: mask
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
    init=.false.
    nullify(xb1,xb2)
end if
if (associated(xb1)) deallocate(xb1)
if (associated(xb2)) deallocate(xb2)
dx=(x2-x1)/n                          Determine the spacing appropriate to the mesh.
x=x1+dx*arth(0,1,n+1)
do i=0,n                              Evaluate the function at the mesh points.
    f(i)=func(x(i))
end do
mask=f(1:n)*f(0:n-1) <= 0.0           Record where the sign changes occur.
nb=count(mask)                        Number of sign changes.
allocate(xb1(nb),xb2(nb))
xb1(1:nb)=pack(x(0:n-1),mask)         Store the bounds of each bracket.
xb2(1:nb)=pack(x(1:n),mask)
END SUBROUTINE zbrak
```

This routine shows how to return arrays `xb1` and `xb2` whose size is not known in advance. The coding is explained in the subsection on pointers in §21.5.

⋆ ⋆ ⋆

```
FUNCTION rtbis(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtbis
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
```

```
INTEGER(I4B), PARAMETER :: MAXIT=40
```
Using bisection, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `rtbis`, will be refined until its accuracy is $\pm$`xacc`.
Parameter: `MAXIT` is the maximum allowed number of bisections.
```
INTEGER(I4B) :: j
REAL(SP) :: dx,f,fmid,xmid
fmid=func(x2)
f=func(x1)
if (f*fmid >= 0.0) call nrerror('rtbis: root must be bracketed')
if (f < 0.0) then           Orient the search so that f>0 lies at x+dx.
    rtbis=x1
    dx=x2-x1
else
    rtbis=x2
    dx=x1-x2
end if
do j=1,MAXIT                Bisection loop.
    dx=dx*0.5_sp
    xmid=rtbis+dx
    fmid=func(xmid)
    if (fmid <= 0.0) rtbis=xmid
    if (abs(dx) < xacc .or. fmid == 0.0) RETURN
end do
call nrerror('rtbis: too many bisections')
END FUNCTION rtbis
```

$\star$ $\star$ $\star$

```
FUNCTION rtflsp(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror,swap
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtflsp
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=30
```
Using the false position method, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `rtflsp`, is refined until its accuracy is $\pm$`xacc`.
Parameter: `MAXIT` is the maximum allowed number of iterations.
```
INTEGER(I4B) :: j
REAL(SP) :: del,dx,f,fh,fl,xh,xl
fl=func(x1)
fh=func(x2)                                 Be sure the interval brackets a root.
if ((fl > 0.0 .and. fh > 0.0) .or. &
    (fl < 0.0 .and. fh < 0.0)) call &
    nrerror('rtflsp: root must be bracketed between arguments')
if (fl < 0.0) then                          Identify the limits so that xl corresponds to
    xl=x1                                       the low side.
    xh=x2
else
    xl=x2
    xh=x1
    call swap(fl,fh)
end if
dx=xh-xl
```

```
do j=1,MAXIT                                    False position loop.
    rtflsp=xl+dx*fl/(fl-fh)                     Increment with respect to latest value.
    f=func(rtflsp)
    if (f < 0.0) then                           Replace appropriate limit.
        del=xl-rtflsp
        xl=rtflsp
        fl=f
    else
        del=xh-rtflsp
        xh=rtflsp
        fh=f
    end if
    dx=xh-xl
    if (abs(del) < xacc .or. f == 0.0) RETURN       Convergence.
end do
call nrerror('rtflsp exceed maximum iterations')
END FUNCTION rtflsp
```

⋆ ⋆ ⋆

```
FUNCTION rtsec(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror,swap
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsec
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=30
    Using the secant method, find the root of a function func thought to lie between x1 and
    x2. The root, returned as rtsec, is refined until its accuracy is ±xacc.
    Parameter: MAXIT is the maximum allowed number of iterations.
INTEGER(I4B) :: j
REAL(SP) :: dx,f,fl,xl
fl=func(x1)
f=func(x2)
if (abs(fl) < abs(f)) then                      Pick the bound with the smaller function value
    rtsec=x1                                        as the most recent guess.
    xl=x2
    call swap(fl,f)
else
    xl=x1
    rtsec=x2
end if
do j=1,MAXIT                                    Secant loop.
    dx=(xl-rtsec)*f/(f-fl)                      Increment with respect to latest value.
    xl=rtsec
    fl=f
    rtsec=rtsec+dx
    f=func(rtsec)
    if (abs(dx) < xacc .or. f == 0.0) RETURN      Convergence.
end do
call nrerror('rtsec: exceed maximum iterations')
END FUNCTION rtsec
```

⋆ ⋆ ⋆

```
FUNCTION zriddr(func,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: zriddr
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=60
```

Using Ridders' method, return the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `zriddr`, will be refined to an approximate accuracy `xacc`.

```
REAL(SP), PARAMETER :: UNUSED=-1.11e30_sp
INTEGER(I4B) :: j
REAL(SP) :: fh,fl,fm,fnew,s,xh,xl,xm,xnew
fl=func(x1)
fh=func(x2)
if ((fl > 0.0 .and. fh < 0.0) .or. (fl < 0.0 .and. fh > 0.0)) then
    xl=x1
    xh=x2
    zriddr=UNUSED                                  Any highly unlikely value, to simplify logic
    do j=1,MAXIT                                       below.
        xm=0.5_sp*(xl+xh)
        fm=func(xm)                                First of two function evaluations per it-
        s=sqrt(fm**2-fl*fh)                            eration.
        if (s == 0.0) RETURN
        xnew=xm+(xm-xl)*(sign(1.0_sp,fl-fh)*fm/s)      Updating formula.
        if (abs(xnew-zriddr) <= xacc) RETURN
        zriddr=xnew
        fnew=func(zriddr)                          Second of two function evaluations per
        if (fnew == 0.0) RETURN                        iteration.
        if (sign(fm,fnew) /= fm) then              Bookkeeping to keep the root bracketed
            xl=xm                                      on next iteration.
            fl=fm
            xh=zriddr
            fh=fnew
        else if (sign(fl,fnew) /= fl) then
            xh=zriddr
            fh=fnew
        else if (sign(fh,fnew) /= fh) then
            xl=zriddr
            fl=fnew
        else
            call nrerror('zriddr: never get here')
        end if
        if (abs(xh-xl) <= xacc) RETURN
    end do
    call nrerror('zriddr: exceeded maximum iterations')
else if (fl == 0.0) then
    zriddr=x1
else if (fh == 0.0) then
    zriddr=x2
else
    call nrerror('zriddr: root must be bracketed')
end if
END FUNCTION zriddr
```

⋆ ⋆ ⋆

```
FUNCTION zbrent(func,x1,x2,tol)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,tol
REAL(SP) :: zbrent
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: EPS=epsilon(x1)
```

Using Brent's method, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `zbrent`, will be refined until its accuracy is `tol`.

Parameters: Maximum allowed number of iterations, and machine floating-point precision.

```
INTEGER(I4B) :: iter
REAL(SP) :: a,b,c,d,e,fa,fb,fc,p,q,r,s,tol1,xm
a=x1
b=x2
fa=func(a)
fb=func(b)
if ((fa > 0.0 .and. fb > 0.0) .or. (fa < 0.0 .and. fb < 0.0)) &
    call nrerror('root must be bracketed for zbrent')
c=b
fc=fb
do iter=1,ITMAX
    if ((fb > 0.0 .and. fc > 0.0) .or. (fb < 0.0 .and. fc < 0.0)) then
        c=a                                     Rename a, b, c and adjust bounding in-
        fc=fa                                       terval d.
        d=b-a
        e=d
    end if
    if (abs(fc) < abs(fb)) then
        a=b
        b=c
        c=a
        fa=fb
        fb=fc
        fc=fa
    end if
    tol1=2.0_sp*EPS*abs(b)+0.5_sp*tol           Convergence check.
    xm=0.5_sp*(c-b)
    if (abs(xm) <= tol1 .or. fb == 0.0) then
        zbrent=b
        RETURN
    end if
    if (abs(e) >= tol1 .and. abs(fa) > abs(fb)) then
        s=fb/fa                                 Attempt inverse quadratic interpolation.
        if (a == c) then
            p=2.0_sp*xm*s
            q=1.0_sp-s
        else
            q=fa/fc
            r=fb/fc
            p=s*(2.0_sp*xm*q*(q-r)-(b-a)*(r-1.0_sp))
            q=(q-1.0_sp)*(r-1.0_sp)*(s-1.0_sp)
        end if
        if (p > 0.0) q=-q                       Check whether in bounds.
        p=abs(p)
        if (2.0_sp*p < min(3.0_sp*xm*q-abs(tol1*q),abs(e*q))) then
            e=d                                 Accept interpolation.
```

```
            d=p/q
        else
            d=xm                                    Interpolation failed; use bisection.
            e=d
        end if
    else                                            Bounds decreasing too slowly; use bisec-
        d=xm                                            tion.
        e=d
    end if
    a=b                                             Move last best guess to a.
    fa=fb
    b=b+merge(d,sign(tol1,xm), abs(d) > tol1 )      Evaluate new trial root.
    fb=func(b)
end do
call nrerror('zbrent: exceeded maximum iterations')
zbrent=b
END FUNCTION zbrent
```

`REAL(SP), PARAMETER :: EPS=epsilon(x1)` The routine `zbrent` works best when EPS is *exactly* the machine precision. The Fortran 90 intrinsic function `epsilon` allows us to code this in a portable fashion.

```
FUNCTION rtnewt(funcd,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtnewt
INTERFACE
    SUBROUTINE funcd(x,fval,fderiv)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: fval,fderiv
    END SUBROUTINE funcd
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=20
    Using the Newton-Raphson method, find the root of a function known to lie in the interval
    [x1, x2]. The root rtnewt will be refined until its accuracy is known within ±xacc. funcd
    is a user-supplied subroutine that returns both the function value and the first derivative of
    the function.
    Parameter: MAXIT is the maximum number of iterations.
INTEGER(I4B) :: j
REAL(SP) :: df,dx,f
rtnewt=0.5_sp*(x1+x2)                       Initial guess.
do j=1,MAXIT
    call funcd(rtnewt,f,df)
    dx=f/df
    rtnewt=rtnewt-dx
    if ((x1-rtnewt)*(rtnewt-x2) < 0.0)&
        call nrerror('rtnewt: values jumped out of brackets')
    if (abs(dx) < xacc) RETURN              Convergence.
end do
call nrerror('rtnewt exceeded maximum iterations')
END FUNCTION rtnewt
```

⋆ ⋆ ⋆

```
FUNCTION rtsafe(funcd,x1,x2,xacc)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,xacc
REAL(SP) :: rtsafe
INTERFACE
    SUBROUTINE funcd(x,fval,fderiv)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: fval,fderiv
    END SUBROUTINE funcd
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXIT=100
```

Using a combination of Newton-Raphson and bisection, find the root of a function bracketed between x1 and x2. The root, returned as the function value rtsafe, will be refined until its accuracy is known within ±xacc. funcd is a user-supplied subroutine that returns both the function value and the first derivative of the function.

Parameter: MAXIT is the maximum allowed number of iterations.

```
INTEGER(I4B) :: j
REAL(SP) :: df,dx,dxold,f,fh,fl,temp,xh,xl
call funcd(x1,fl,df)
call funcd(x2,fh,df)
if ((fl > 0.0 .and. fh > 0.0) .or. &
    (fl < 0.0 .and. fh < 0.0)) &
    call nrerror('root must be bracketed in rtsafe')
if (fl == 0.0) then
    rtsafe=x1
    RETURN
else if (fh == 0.0) then
    rtsafe=x2
    RETURN
else if (fl < 0.0) then                 Orient the search so that f(xl) < 0.
    xl=x1
    xh=x2
else
    xh=x1
    xl=x2
end if
rtsafe=0.5_sp*(x1+x2)                   Initialize the guess for root,
dxold=abs(x2-x1)                        the "stepsize before last,"
dx=dxold                                and the last step.
call funcd(rtsafe,f,df)
do j=1,MAXIT                            Loop over allowed iterations.
    if (((rtsafe-xh)*df-f)*((rtsafe-xl)*df-f) > 0.0 .or. &
        abs(2.0_sp*f) > abs(dxold*df) ) then
          Bisect if Newton out of range, or not decreasing fast enough.
        dxold=dx
        dx=0.5_sp*(xh-xl)
        rtsafe=xl+dx
        if (xl == rtsafe) RETURN        Change in root is negligible.
    else                                Newton step acceptable. Take it.
        dxold=dx
        dx=f/df
        temp=rtsafe
        rtsafe=rtsafe-dx
        if (temp == rtsafe) RETURN
    end if
    if (abs(dx) < xacc) RETURN          Convergence criterion.
    call funcd(rtsafe,f,df)             One new function evaluation per iteration.
    if (f < 0.0) then                   Maintain the bracket on the root.
        xl=rtsafe
    else
        xh=rtsafe
```

```
    end if
end do
call nrerror('rtsafe: exceeded maximum iterations')
END FUNCTION rtsafe
```

⋆ ⋆ ⋆

```
SUBROUTINE laguer(a,x,its)
USE nrtype; USE nrutil, ONLY : nrerror,poly,poly_term
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: its
COMPLEX(SPC), INTENT(INOUT) :: x
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
REAL(SP), PARAMETER :: EPS=epsilon(1.0_sp)
INTEGER(I4B), PARAMETER :: MR=8,MT=10,MAXIT=MT*MR
```

Given an array of $M+1$ complex coefficients a of the polynomial $\sum_{i=1}^{M+1} \mathtt{a}(i)x^{i-1}$, and given a complex value x, this routine improves x by Laguerre's method until it converges, within the achievable roundoff limit, to a root of the given polynomial. The number of iterations taken is returned as its.

Parameters: EPS is the estimated fractional roundoff error. We try to break (rare) limit cycles with MR different fractional values, once every MT steps, for MAXIT total allowed iterations.

```
INTEGER(I4B) :: iter,m
REAL(SP) :: abx,abp,abm,err
COMPLEX(SPC) :: dx,x1,f,g,h,sq,gp,gm,g2
COMPLEX(SPC), DIMENSION(size(a)) :: b,d
REAL(SP), DIMENSION(MR) :: frac = &
    (/ 0.5_sp,0.25_sp,0.75_sp,0.13_sp,0.38_sp,0.62_sp,0.88_sp,1.0_sp /)
```

Fractions used to break a limit cycle.

```
m=size(a)-1
do iter=1,MAXIT                                  Loop over iterations up to allowed maximum.
    its=iter
    abx=abs(x)
    b(m+1:1:-1)=poly_term(a(m+1:1:-1),x)          Efficient computation of the polynomial
    d(m:1:-1)=poly_term(b(m+1:2:-1),x)                and its first two derivatives.
    f=poly(x,d(2:m))
    err=EPS*poly(abx,abs(b(1:m+1)))               Esimate of roundoff in evaluating polynomial.
    if (abs(b(1)) <= err) RETURN                  We are on the root.
    g=d(1)/b(1)                                   The generic case: Use Laguerre's formula.
    g2=g*g
    h=g2-2.0_sp*f/b(1)
    sq=sqrt((m-1)*(m*h-g2))
    gp=g+sq
    gm=g-sq
    abp=abs(gp)
    abm=abs(gm)
    if (abp < abm) gp=gm
    if (max(abp,abm) > 0.0) then
        dx=m/gp
    else
        dx=exp(cmplx(log(1.0_sp+abx),iter,kind=spc))
    end if
    x1=x-dx
    if (x == x1) RETURN                           Converged.
    if (mod(iter,MT) /= 0) then
        x=x1
    else                                          Every so often we take a fractional step, to
        x=x-dx*frac(iter/MT)                          break any limit cycle (itself a rare occur-
    end if                                            rence).
end do
call nrerror('laguer: too many iterations')
```

Very unusual — can occur only for complex roots. Try a different starting guess for the root.

```
END SUBROUTINE laguer
```

`b(m+1:1:-1)=poly_term...f=poly(x,d(2:m))` The `poly_term` function in `nrutil` tabulates the partial sums of a polynomial, while `poly` evaluates the polynomial at `x`. In this example, we use `poly_term` on the coefficient array in reverse order, so that the value of the polynomial ends up in `b(1)` and the value of its first derivative in `d(1)`.

`dx=exp(cmplx(log(1.0_sp+abx),iter,kind=spc))` The intrinsic function `cmplx` returns a quantity of type default complex unless the `kind` argument is present. To facilitate converting our routines from single to double precision, we always include the `kind` argument explicitly so that when you redefine `spc` in `nrtype` to be double-precision complex the conversions are carried out correctly.

⋆ ⋆ ⋆

```
SUBROUTINE zroots(a,roots,polish)
USE nrtype; USE nrutil, ONLY : assert_eq,poly_term
USE nr, ONLY : laguer,indexx
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: roots
LOGICAL(LGT), INTENT(IN) :: polish
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Given the array of $M+1$ complex coefficients `a` of the polynomial $\sum_{i=1}^{M+1} \mathtt{a}(i)x^{i-1}$, this routine successively calls `laguer` and finds all M complex `roots`. The logical variable `polish` should be input as `.true.` if polishing (also by Laguerre's method) is desired, `.false.` if the roots will be subsequently polished by other means.

Parameter: EPS is a small number.

```
INTEGER(I4B) :: j,its,m
INTEGER(I4B), DIMENSION(size(roots)) :: indx
COMPLEX(SPC) :: x
COMPLEX(SPC), DIMENSION(size(a)) :: ad
m=assert_eq(size(roots),size(a)-1,'zroots')
ad(:)=a(:)                                  Copy of coefficients for successive deflation.
do j=m,1,-1                                 Loop over each root to be found.
    x=cmplx(0.0_sp,kind=spc)
      Start at zero to favor convergence to smallest remaining root.
    call laguer(ad(1:j+1),x,its)            Find the root.
    if (abs(aimag(x)) <= 2.0_sp*EPS**2*abs(real(x))) &
        x=cmplx(real(x),kind=spc)
    roots(j)=x
    ad(j:1:-1)=poly_term(ad(j+1:2:-1),x)        Forward deflation.
end do
if (polish) then
    do j=1,m                                Polish the roots using the undeflated coeffi-
        call laguer(a(:),roots(j),its)          cients.
    end do
end if
call indexx(real(roots),indx)               Sort roots by their real parts.
roots=roots(indx)
END SUBROUTINE zroots
```

`x=cmplx(0.0_sp,kind=spc)...x=cmplx(real(x),kind=spc)` See the discussion of why we include `kind=spc` just above. Note that while `real(x)` returns type default real if `x` is integer or real, it returns single or double precision correctly if `x` is complex.

⋆ ⋆ ⋆

```
SUBROUTINE zrhqr(a,rtr,rti)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : balanc,hqr,indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: rtr,rti
```

Find all the roots of a polynomial with real coefficients, $\sum_{i=1}^{M+1} \mathtt{a}(i)x^{i-1}$, given the array of $M+1$ coefficients a. The method is to construct an upper Hessenberg matrix whose eigenvalues are the desired roots, and then use the routines balanc and hqr. The real and imaginary parts of the M roots are returned in rtr and rti, respectively.

```
INTEGER(I4B) :: k,m
INTEGER(I4B), DIMENSION(size(rtr)) :: indx
REAL(SP), DIMENSION(size(a)-1,size(a)-1) :: hess
m=assert_eq(size(rtr),size(rti),size(a)-1,'zrhqr')
if (a(m+1) == 0.0) call &
    nrerror('zrhqr: Last value of array a must not be 0')
hess(1,:)=-a(m:1:-1)/a(m+1)          Construct the matrix.
hess(2:m,:)=0.0
do k=1,m-1
    hess(k+1,k)=1.0
end do
call balanc(hess)                    Find its eigenvalues.
call hqr(hess,rtr,rti)
call indexx(rtr,indx)                Sort roots by their real parts.
rtr=rtr(indx)
rti=rti(indx)
END SUBROUTINE zrhqr
```

⋆ ⋆ ⋆

```
SUBROUTINE qroot(p,b,c,eps)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : poldiv
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: p
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: eps
INTEGER(I4B), PARAMETER :: ITMAX=20
REAL(SP), PARAMETER :: TINY=1.0e-6_sp
```

Given an array of N coefficients p of a polynomial of degree $N-1$, and trial values for the coefficients of a quadratic factor $x^2 + \mathtt{b}x + \mathtt{c}$, improve the solution until the coefficients b,c change by less than eps. The routine poldiv of §5.3 is used.

Parameters: ITMAX is the maximum number of iterations, TINY is a small number.

```
INTEGER(I4B) :: iter,n
REAL(SP) :: delb,delc,div,r,rb,rc,s,sb,sc
REAL(SP), DIMENSION(3) :: d
REAL(SP), DIMENSION(size(p)) :: q,qq,rem
n=size(p)
d(3)=1.0
do iter=1,ITMAX
    d(2)=b
    d(1)=c
    call poldiv(p,d,q,rem)
    s=rem(1)                         First division gives r,s.
    r=rem(2)
    call poldiv(q(1:n-1),d(:),qq(1:n-1),rem(1:n-1))
    sc=-rem(1)                       Second division gives partial r,s with respect
    rc=-rem(2)                           to c.
    sb=-c*rc
    rb=sc-b*rc
    div=1.0_sp/(sb*rc-sc*rb)         Solve 2x2 equation.
```

```
    delb=(r*sc-s*rc)*div
    delc=(-r*sb+s*rb)*div
    b=b+delb
    c=c+delc
    if ((abs(delb) <= eps*abs(b) .or. abs(b) < TINY) .and. &
        (abs(delc) <= eps*abs(c) .or. abs(c) < TINY)) RETURN   Coefficients converged.
end do
call nrerror('qroot: too many iterations')
END SUBROUTINE qroot
```

⋆ ⋆ ⋆

```
SUBROUTINE mnewt(ntrial,x,tolx,tolf,usrfun)
USE nrtype
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ntrial
REAL(SP), INTENT(IN) :: tolx,tolf
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
INTERFACE
    SUBROUTINE usrfun(x,fvec,fjac)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: fvec
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: fjac
    END SUBROUTINE usrfun
END INTERFACE
```

Given an initial guess `x` for a root in N dimensions, take `ntrial` Newton-Raphson steps to improve the root. Stop if the root converges in either summed absolute variable increments `tolx` or summed absolute function values `tolf`.

```
INTEGER(I4B) :: i
INTEGER(I4B), DIMENSION(size(x)) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(size(x)) :: fvec,p
REAL(SP), DIMENSION(size(x),size(x)) :: fjac
do i=1,ntrial
    call usrfun(x,fvec,fjac)
```

User subroutine supplies function values at `x` in `fvec` and Jacobian matrix in `fjac`.

```
    if (sum(abs(fvec)) <= tolf) RETURN        Check function convergence.
    p=-fvec                                   Right-hand side of linear equations.
    call ludcmp(fjac,indx,d)                  Solve linear equations using LU decom-
    call lubksb(fjac,indx,p)                      position.
    x=x+p                                     Update solution.
    if (sum(abs(p)) <= tolx) RETURN           Check root convergence.
end do
END SUBROUTINE mnewt
```

⋆ ⋆ ⋆

```
SUBROUTINE lnsrch(xold,fold,g,p,x,f,stpmax,check,func)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror,vabs
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xold,g
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
REAL(SP), INTENT(IN) :: fold,stpmax
REAL(SP), DIMENSION(:), INTENT(OUT) :: x
REAL(SP), INTENT(OUT) :: f
LOGICAL(LGT), INTENT(OUT) :: check
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP) :: func
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: ALF=1.0e-4_sp,TOLX=epsilon(x)
```

Given an N-dimensional point xold, the value of the function and gradient there, fold and g, and a direction p, finds a new point x along the direction p from xold where the function func has decreased "sufficiently." xold, g, p, and x are all arrays of length N. The new function value is returned in f. stpmax is an input quantity that limits the length of the steps so that you do not try to evaluate the function in regions where it is undefined or subject to overflow. p is usually the Newton direction. The output quantity check is false on a normal exit. It is true when x is too close to xold. In a minimization algorithm, this usually signals convergence and can be ignored. However, in a zero-finding algorithm the calling program should check whether the convergence is spurious.

Parameters: ALF ensures sufficient decrease in function value; TOLX is the convergence criterion on Δx.

```
INTEGER(I4B) :: ndum
REAL(SP) :: a,alam,alam2,alamin,b,disc,f2,pabs,rhs1,rhs2,slope,tmplam
ndum=assert_eq(size(g),size(p),size(x),size(xold),'lnsrch')
check=.false.
pabs=vabs(p(:))
if (pabs > stpmax) p(:)=p(:)*stpmax/pabs          Scale if attempted step is too big.
slope=dot_product(g,p)
if (slope >= 0.0) call nrerror('roundoff problem in lnsrch')
alamin=TOLX/maxval(abs(p(:))/max(abs(xold(:)),1.0_sp))      Compute λmin.
alam=1.0                                          Always try full Newton step first.
do                                                Start of iteration loop.
    x(:)=xold(:)+alam*p(:)
    f=func(x)
    if (alam < alamin) then                       Convergence on Δx. For zero finding, the calling program should verify the convergence.
        x(:)=xold(:)
        check=.true.
        RETURN
    else if (f <= fold+ALF*alam*slope) then       Sufficient function decrease.
        RETURN
    else                                          Backtrack.
        if (alam == 1.0) then                     First time.
            tmplam=-slope/(2.0_sp*(f-fold-slope))
        else                                      Subsequent backtracks.
            rhs1=f-fold-alam*slope
            rhs2=f2-fold-alam2*slope
            a=(rhs1/alam**2-rhs2/alam2**2)/(alam-alam2)
            b=(-alam2*rhs1/alam**2+alam*rhs2/alam2**2)/&
                (alam-alam2)
            if (a == 0.0) then
                tmplam=-slope/(2.0_sp*b)
            else
                disc=b*b-3.0_sp*a*slope
                if (disc < 0.0) then
                    tmplam=0.5_sp*alam
                else if (b <= 0.0) then
```

```
                tmplam=(-b+sqrt(disc))/(3.0_sp*a)
            else
                tmplam=-slope/(b+sqrt(disc))
            end if
        end if
        if (tmplam > 0.5_sp*alam) tmplam=0.5_sp*alam    λ ≤ 0.5λ1.
    end if
    alam2=alam
    f2=f
    alam=max(tmplam,0.1_sp*alam)                        λ ≥ 0.1λ1.
end do                                                  Try again.
END SUBROUTINE lnsrch
```

```
SUBROUTINE newt(x,check)
USE nrtype; USE nrutil, ONLY : nrerror,vabs
USE nr, ONLY : fdjac,lnsrch,lubksb,ludcmp
USE fminln                                  Communicates with fmin.
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
INTEGER(I4B), PARAMETER :: MAXITS=200
REAL(SP), PARAMETER :: TOLF=1.0e-4_sp,TOLMIN=1.0e-6_sp,TOLX=epsilon(x),&
    STPMX=100.0
```

Given an initial guess x for a root in N dimensions, find the root by a globally convergent Newton's method. The length N vector of functions to be zeroed, called fvec in the routine below, is returned by a user-supplied routine that *must* be called funcv and have the declaration FUNCTION funcv(x). The output quantity check is false on a normal return and true if the routine has converged to a local minimum of the function fmin defined below. In this case try restarting from a different initial guess.

Parameters: MAXITS is the maximum number of iterations; TOLF sets the convergence criterion on function values; TOLMIN sets the criterion for deciding whether spurious convergence to a minimum of fmin has occurred; TOLX is the convergence criterion on $\delta\mathbf{x}$; STPMX is the scaled maximum step length allowed in line searches.

```
INTEGER(I4B) :: its
INTEGER(I4B), DIMENSION(size(x)) :: indx
REAL(SP) :: d,f,fold,stpmax
REAL(SP), DIMENSION(size(x)) :: g,p,xold
REAL(SP), DIMENSION(size(x)), TARGET :: fvec
REAL(SP), DIMENSION(size(x),size(x)) :: fjac
fmin_fvecp=>fvec
f=fmin(x)                                   fvec is also computed by this call.
if (maxval(abs(fvec(:))) < 0.01_sp*TOLF) then       Test for initial guess being a root.
    check=.false.                                       Use more stringent test than
    RETURN                                              simply TOLF.
end if
stpmax=STPMX*max(vabs(x(:)),real(size(x),sp))       Calculate stpmax for line searches.
do its=1,MAXITS                             Start of iteration loop.
    call fdjac(x,fvec,fjac)
```

If analytic Jacobian is available, you can replace the routine fdjac below with your own routine.

```
    g(:)=matmul(fvec(:),fjac(:,:))          Compute ∇f for the line search.
    xold(:)=x(:)                            Store x,
    fold=f                                  and f.
    p(:)=-fvec(:)                           Right-hand side for linear equations.
    call ludcmp(fjac,indx,d)                Solve linear equations by LU decomposition.
    call lubksb(fjac,indx,p)
    call lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin)
```

lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.

```
    if (maxval(abs(fvec(:))) < TOLF) then       Test for convergence on function val-
        check=.false.                               ues.
        RETURN
```

```
    end if
    if (check) then                          Check for gradient of f zero, i.e., spurious
        check=(maxval(abs(g(:))*max(abs(x(:)),1.0_sp) / &      convergence.
            max(f,0.5_sp*size(x))) < TOLMIN)
        RETURN                               Test for convergence on δx.
    end if
    if (maxval(abs(x(:)-xold(:))/max(abs(x(:)),1.0_sp)) < TOLX) &
        RETURN
end do
call nrerror('MAXITS exceeded in newt')
END SUBROUTINE newt
```

f90 USE fminln Here we have an example of how to pass an array fvec to a function fmin without making it an argument of fmin. In the language of §21.5, we are using Method 2: We define a pointer fmin_fvecp in the module fminln:

```
REAL(SP), DIMENSION(:), POINTER :: fmin_fvecp
```

fvec itself is declared as an automatic array of the appropriate size in newt:

```
REAL(SP), DIMENSION(size(x)), TARGET :: fvec
```

On entry into newt, the pointer is associated:

```
fmin_fvecp=>fvec
```

The pointer is then used in fmin as a synonym for fvec. If you are sufficiently paranoid, you can test whether fmin_fvecp has in fact been associated on entry into fmin. Heeding our admonition always to deallocate memory when it no longer is needed, you may ask where the deallocation takes place in this example. Answer: On exit from newt, the automatic array fvec is automatically freed.

The Method 1 way of setting up this task is to declare an allocatable array in the module:

```
REAL(SP), DIMENSION(:), ALLOCATABLE :: fvec
```

On entry into newt we allocate it appropriately:

```
allocate(fvec,size(x))
```

and it can now be used in both newt and fmin. Of course, we must remember to deallocate explicitly fvec on exit from newt. If we forget, all kinds of bad things would happen on a second call to newt. The status of fvec on the first return from newt becomes undefined. The status cannot be tested with if(allocated(...)), and fvec may not be referenced in any way. If we tried to guard against this by adding the SAVE attribute to the declaration of fvec, then we would generate an error from trying to allocate an already-allocated array.

```
SUBROUTINE fdjac(x,fvec,df)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: fvec
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: df
INTERFACE
```

```
    FUNCTION funcv(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funcv
    END FUNCTION funcv
END INTERFACE
REAL(SP), PARAMETER :: EPS=1.0e-4_sp
```

Computes forward-difference approximation to Jacobian. On input, `x` is the point at which the Jacobian is to be evaluated, and `fvec` is the vector of function values at the point, both arrays of length N. `df` is the $N \times N$ output Jacobian. `FUNCTION funcv(x)` is a fixed-name, user-supplied routine that returns the vector of functions at `x`.

Parameter: `EPS` is the approximate square root of the machine precision.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(x)) :: xsav,xph,h
n=assert_eq(size(x),size(fvec),size(df,1),size(df,2),'fdjac')
xsav=x
h=EPS*abs(xsav)
where (h == 0.0) h=EPS
xph=xsav+h                                  Trick to reduce finite precision error.
h=xph-xsav
do j=1,n
    x(j)=xph(j)
    df(:,j)=(funcv(x)-fvec(:))/h(j)         Forward difference formula.
    x(j)=xsav(j)
end do
END SUBROUTINE fdjac
```

```
MODULE fminln
USE nrtype; USE nrutil, ONLY : nrerror
REAL(SP), DIMENSION(:), POINTER :: fmin_fvecp
CONTAINS

FUNCTION fmin(x)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x
REAL(SP) :: fmin
```

Returns $f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F}$ at `x`. `FUNCTION funcv(x)` is a fixed-name, user-supplied routine that returns the vector of functions at `x`. The pointer `fmin_vecp` communicates the function values back to `newt`.

```
INTERFACE
    FUNCTION funcv(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funcv
    END FUNCTION funcv
END INTERFACE
if (.not. associated(fmin_fvecp)) call &
    nrerror('fmin: problem with pointer for returned values')
fmin_fvecp=funcv(x)
fmin=0.5_sp*dot_product(fmin_fvecp,fmin_fvecp)
END FUNCTION fmin
END MODULE fminln
```

⋆ ⋆ ⋆

```
SUBROUTINE broydn(x,check)
USE nrtype; USE nrutil, ONLY : get_diag,lower_triangle,nrerror,&
    outerprod,put_diag,unit_matrix,vabs
USE nr, ONLY : fdjac,lnsrch,qrdcmp,qrupdt,rsolv
USE fminln                                          Communicates with fmin.
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
LOGICAL(LGT), INTENT(OUT) :: check
INTEGER(I4B), PARAMETER :: MAXITS=200
REAL(SP), PARAMETER :: EPS=epsilon(x),TOLF=1.0e-4_sp,TOLMIN=1.0e-6_sp,&
    TOLX=EPS,STPMX=100.0
    Given an initial guess x for a root in N dimensions, find the root by Broyden's method
    embedded in a globally convergent strategy. The length N vector of functions to be ze-
    roed, called fvec in the routine below, is returned by a user-supplied routine that must be
    called funcv and have the declaration FUNCTION funcv(x). The subroutine fdjac and
    the function fmin from newt are used. The output quantity check is false on a normal
    return and true if the routine has converged to a local minimum of the function fmin or if
    Broyden's method can make no further progress. In this case try restarting from a different
    initial guess.
    Parameters: MAXITS is the maximum number of iterations; EPS is the machine precision;
    TOLF sets the convergence criterion on function values; TOLMIN sets the criterion for de-
    ciding whether spurious convergence to a minimum of fmin has occurred; TOLX is the
    convergence criterion on δx; STPMX is the scaled maximum step length allowed in line
    searches.
INTEGER(I4B) :: i,its,k,n
REAL(SP) :: f,fold,stpmax
REAL(SP), DIMENSION(size(x)), TARGET :: fvec
REAL(SP), DIMENSION(size(x)) :: c,d,fvcold,g,p,s,t,w,xold
REAL(SP), DIMENSION(size(x),size(x)) :: qt,r
LOGICAL :: restrt,sing
fmin_fvecp=>fvec
n=size(x)
f=fmin(x)                                           fvec is also computed by this call.
if (maxval(abs(fvec(:))) < 0.01_sp*TOLF) then       Test for initial guess being a root.
    check=.false.                                       Use more stringent test than
    RETURN                                              simply TOLF.
end if
stpmax=STPMX*max(vabs(x(:)),real(n,sp))             Calculate stpmax for line searches.
restrt=.true.                                       Ensure initial Jacobian gets computed.
do its=1,MAXITS                                     Start of iteration loop.
    if (restrt) then
        call fdjac(x,fvec,r)                        Initialize or reinitialize Jacobian in r.
        call qrdcmp(r,c,d,sing)                     QR decomposition of Jacobian.
        if (sing) call nrerror('singular Jacobian in broydn')
        call unit_matrix(qt)                        Form Q^T explicitly.
        do k=1,n-1
            if (c(k) /= 0.0) then
                qt(k:n,:)=qt(k:n,:)-outerprod(r(k:n,k),&
                    matmul(r(k:n,k),qt(k:n,:)))/c(k)
            end if
        end do
        where (lower_triangle(n,n)) r(:,:)=0.0
        call put_diag(d(:),r(:,:))                  Form R explicitly.
    else                                            Carry out Broyden update.
        s(:)=x(:)-xold(:)                           s = δx.
        do i=1,n                                    t = R · s.
            t(i)=dot_product(r(i,i:n),s(i:n))
        end do
        w(:)=fvec(:)-fvcold(:)-matmul(t(:),qt(:,:))     w = δF − B · s.
        where (abs(w(:)) < EPS*(abs(fvec(:))+abs(fvcold(:)))) &
            w(:)=0.0                                Don't update with noisy components of
        if (any(w(:) /= 0.0)) then                      w.
            t(:)=matmul(qt(:,:),w(:))               t = Q^T · w.
            s(:)=s(:)/dot_product(s,s)              Store s/(s · s) in s.
```

```
            call qrupdt(r,qt,t,s)                Update R and Q^T.
            d(:)=get_diag(r(:,:))                Diagonal of R stored in d.
            if (any(d(:) == 0.0)) &
                call nrerror('r singular in broydn')
        end if
    end if
    p(:)=-matmul(qt(:,:),fvec(:))                r.h.s. for linear equations is −Q^T · F.
    do i=1,n                                     Compute ∇f ≈ (Q · R)^T · F for the line
        g(i)=-dot_product(r(1:i,i),p(1:i))           search.
    end do
    xold(:)=x(:)                                 Store x, F, and f.
    fvcold(:)=fvec(:)
    fold=f
    call rsolv(r,d,p)                            Solve linear equations.
    call lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin)
      lnsrch returns new x and f. It also calculates fvec at the new x when it calls fmin.
    if (maxval(abs(fvec(:))) < TOLF) then            Test for convergence on function val-
        check=.false.                                    ues.
        RETURN
    end if
    if (check) then                              True if line search failed to find a new
        if (restrt .or. maxval(abs(g(:))*max(abs(x(:)), &              x.
            1.0_sp)/max(f,0.5_sp*n)) < TOLMIN) RETURN
              If restrt is true we have failure: We have already tried reinitializing the Jacobian. The other test is for gradient of f zero, i.e., spurious convergence.
        restrt=.true.                            Try reinitializing the Jacobian.
    else                                         Successful step; will use Broyden update
        restrt=.false.                               for next step.
        if (maxval((abs(x(:)-xold(:)))/max(abs(x(:)), &
            1.0_sp)) < TOLX) RETURN              Test for convergence on δx.
    end if
end do
call nrerror('MAXITS exceeded in broydn')
END SUBROUTINE broydn
```

`USE fminln` See discussion for `newt` on p. 1197.

`qt(k:n,:)=...outerprod...matmul` Another example of the coding of equation (22.1.6).

`where (lower_triangle(n,n))...` The `lower_triangle` function in `nrutil` returns a lower triangular logical mask. As used here, the mask is true everywhere in the lower triangle of an `n` × `n` matrix, excluding the diagonal. An optional integer argument `extra` allows additional diagonals to be set to true. With `extra=1` the lower triangle including the diagonal would be true.

`call put_diag(d(:),r(:,:))` This subroutine in `nrutil` sets the diagonal values of the matrix `r` to the values of the vector `d`. It is overloaded so that `d` could be a scalar, in which case the scalar value would be broadcast onto the diagonal of `r`.

Chapter B10. Minimization or Maximization of Functions

```
SUBROUTINE mnbrak(ax,bx,cx,fa,fb,fc,func)
USE nrtype; USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), INTENT(INOUT) :: ax,bx
REAL(SP), INTENT(OUT) :: cx,fa,fb,fc
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: GOLD=1.618034_sp,GLIMIT=100.0_sp,TINY=1.0e-20_sp
```

Given a function `func`, and given distinct initial points `ax` and `bx`, this routine searches in the downhill direction (defined by the function as evaluated at the initial points) and returns new points `ax`, `bx`, `cx` that bracket a minimum of the function. Also returned are the function values at the three points, `fa`, `fb`, and `fc`.

Parameters: `GOLD` is the default ratio by which successive intervals are magnified; `GLIMIT` is the maximum magnification allowed for a parabolic-fit step.

```
REAL(SP) :: fu,q,r,u,ulim
fa=func(ax)
fb=func(bx)
if (fb > fa) then                                   Switch roles of a and b so that we
    call swap(ax,bx)                                    can go downhill in the direction
    call swap(fa,fb)                                    from a to b.
end if
cx=bx+GOLD*(bx-ax)                                  First guess for c.
fc=func(cx)
do                                                  Do-while-loop: Keep returning here
    if (fb < fc) RETURN                                 until we bracket.
```

Compute u by parabolic extrapolation from a, b, c. `TINY` is used to prevent any possible division by zero.

```
    r=(bx-ax)*(fb-fc)
    q=(bx-cx)*(fb-fa)
    u=bx-((bx-cx)*q-(bx-ax)*r)/(2.0_sp*sign(max(abs(q-r),TINY),q-r))
    ulim=bx+GLIMIT*(cx-bx)
```

We won't go farther than this. Test various possibilities:

```
    if ((bx-u)*(u-cx) > 0.0) then                   Parabolic u is between b and c: try
        fu=func(u)                                      it.
        if (fu < fc) then                           Got a minimum between b and c.
            ax=bx
            fa=fb
            bx=u
            fb=fu
            RETURN
        else if (fu > fb) then                      Got a minimum between a and u.
            cx=u
            fc=fu
            RETURN
```

```
        end if
        u=cx+GOLD*(cx-bx)                          Parabolic fit was no use. Use default
        fu=func(u)                                     magnification.
    else if ((cx-u)*(u-ulim) > 0.0) then           Parabolic fit is between c and its al-
        fu=func(u)                                     lowed limit.
        if (fu < fc) then
            bx=cx
            cx=u
            u=cx+GOLD*(cx-bx)
            call shft(fb,fc,fu,func(u))
        end if
    else if ((u-ulim)*(ulim-cx) >= 0.0) then       Limit parabolic u to maximum al-
        u=ulim                                         lowed value.
        fu=func(u)
    else                                           Reject parabolic u, use default mag-
        u=cx+GOLD*(cx-bx)                              nification.
        fu=func(u)
    end if
    call shft(ax,bx,cx,u)
    call shft(fa,fb,fc,fu)                         Eliminate oldest point and continue.
end do
CONTAINS
SUBROUTINE shft(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
c=d
END SUBROUTINE shft
END SUBROUTINE mnbrak
```

f90 `call shft...` There are three places in `mnbrak` where we need to shift four variables around. Rather than repeat code, we make `shft` an internal subroutine, coming after a `CONTAINS` statement. It is invisible to all procedures except `mnbrak`.

⋆ ⋆ ⋆

```
FUNCTION golden(ax,bx,cx,func,tol,xmin)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: golden
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), PARAMETER :: R=0.61803399_sp,C=1.0_sp-R
```

Given a function `func`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `func(bx)` is less than both `func(ax)` and `func(cx)`), this routine performs a golden section search for the minimum, isolating it to a fractional precision of about `tol`. The abscissa of the minimum is returned as `xmin`, and the minimum

function value is returned as golden, the returned function value.
Parameters: The golden ratios.

```
REAL(SP) :: f1,f2,x0,x1,x2,x3
x0=ax                                   At any given time we will keep track of
x3=cx                                       four points, x0,x1,x2,x3.
if (abs(cx-bx) > abs(bx-ax)) then       Make x0 to x1 the smaller segment,
    x1=bx
    x2=bx+C*(cx-bx)                     and fill in the new point to be tried.
else
    x2=bx
    x1=bx-C*(bx-ax)
end if
f1=func(x1)
f2=func(x2)
```

The initial function evaluations. Note that we never need to evaluate the function at the original endpoints.

```
do                                      Do-while-loop: We keep returning here.
    if (abs(x3-x0) <= tol*(abs(x1)+abs(x2))) exit
    if (f2 < f1) then                   One possible outcome,
        call shft3(x0,x1,x2,R*x2+C*x3)  its housekeeping,
        call shft2(f1,f2,func(x2))      and a new function evaluation.
    else                                The other outcome,
        call shft3(x3,x2,x1,R*x1+C*x0)
        call shft2(f2,f1,func(x1))      and its new function evaluation.
    end if
end do                                  Back to see if we are done.
if (f1 < f2) then                       We are done. Output the best of the two
    golden=f1                               current values.
    xmin=x1
else
    golden=f2
    xmin=x2
end if
CONTAINS

SUBROUTINE shft2(a,b,c)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b
REAL(SP), INTENT(IN) :: c
a=b
b=c
END SUBROUTINE shft2

SUBROUTINE shft3(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
c=d
END SUBROUTINE shft3
END FUNCTION golden
```

f90 `call shft3...call shft2...` See discussion of shft for mnbrak on p. 1202.

⋆ ⋆ ⋆

```
FUNCTION brent(ax,bx,cx,func,tol,xmin)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: brent
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: CGOLD=0.3819660_sp,ZEPS=1.0e-3_sp*epsilon(ax)
```

Given a function `func`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `func(bx)` is less than both `func(ax)` and `func(cx)`), this routine isolates the minimum to a fractional precision of about `tol` using Brent's method. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `brent`, the returned function value.

Parameters: Maximum allowed number of iterations; golden ratio; and a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

```
INTEGER(I4B) :: iter
REAL(SP) :: a,b,d,e,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm
a=min(ax,cx)                                  a and b must be in ascending order, though
b=max(ax,cx)                                      the input abscissas need not be.
v=bx                                          Initializations...
w=v
x=v
e=0.0                                         This will be the distance moved on the step
fx=func(x)                                        before last.
fv=fx
fw=fx
do iter=1,ITMAX                               Main program loop.
    xm=0.5_sp*(a+b)
    tol1=tol*abs(x)+ZEPS
    tol2=2.0_sp*tol1
    if (abs(x-xm) <= (tol2-0.5_sp*(b-a))) then        Test for done here.
        xmin=x                                Arrive here ready to exit with best values.
        brent=fx
        RETURN
    end if
    if (abs(e) > tol1) then                   Construct a trial parabolic fit.
        r=(x-w)*(fx-fv)
        q=(x-v)*(fx-fw)
        p=(x-v)*q-(x-w)*r
        q=2.0_sp*(q-r)
        if (q > 0.0) p=-p
        q=abs(q)
        etemp=e
        e=d
        if (abs(p) >= abs(0.5_sp*q*etemp) .or. &
            p <= q*(a-x) .or. p >= q*(b-x)) then
              The above conditions determine the acceptability of the parabolic fit. Here it is
              not o.k., so we take the golden section step into the larger of the two segments.
            e=merge(a-x,b-x, x >= xm )
            d=CGOLD*e
        else                                  Take the parabolic step.
            d=p/q
            u=x+d
            if (u-a < tol2 .or. b-u < tol2) d=sign(tol1,xm-x)
        end if
```

```
    else                                        Take the golden section step into the larger
        e=merge(a-x,b-x, x >= xm )                  of the two segments.
        d=CGOLD*e
    end if
    u=merge(x+d,x+sign(tol1,d), abs(d) >= tol1 )
      Arrive here with d computed either from parabolic fit, or else from golden section.
    fu=func(u)
      This is the one function evaluation per iteration.
    if (fu <= fx) then                          Now we have to decide what to do with our
        if (u >= x) then                            function evaluation. Housekeeping follows:
            a=x
        else
            b=x
        end if
        call shft(v,w,x,u)
        call shft(fv,fw,fx,fu)
    else
        if (u < x) then
            a=u
        else
            b=u
        end if
        if (fu <= fw .or. w == x) then
            v=w
            fv=fw
            w=u
            fw=fu
        else if (fu <= fv .or. v == x .or. v == w) then
            v=u
            fv=fu
        end if
    end if
end do                                          Done with housekeeping. Back for another
call nrerror('brent: exceed maximum iterations')    iteration.
CONTAINS

SUBROUTINE shft(a,b,c,d)
REAL(SP), INTENT(OUT) :: a
REAL(SP), INTENT(INOUT) :: b,c
REAL(SP), INTENT(IN) :: d
a=b
b=c
c=d
END SUBROUTINE shft
END FUNCTION brent
```

⋆ ⋆ ⋆

```
FUNCTION dbrent(ax,bx,cx,func,dfunc,tol,xmin)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(IN) :: ax,bx,cx,tol
REAL(SP), INTENT(OUT) :: xmin
REAL(SP) :: dbrent
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func

    FUNCTION dfunc(x)
```

```
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: dfunc
    END FUNCTION dfunc
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=100
REAL(SP), PARAMETER :: ZEPS=1.0e-3_sp*epsilon(ax)
```

Given a function `func` and its derivative function `dfunc`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` [such that `bx` is between `ax` and `cx`, and `func(bx)` is less than both `func(ax)` and `func(cx)`], this routine isolates the minimum to a fractional precision of about `tol` using a modification of Brent's method that uses derivatives. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `dbrent`, the returned function value.

Parameters: Maximum allowed number of iterations, and a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

```
INTEGER(I4B) :: iter
REAL(SP) :: a,b,d,d1,d2,du,dv,dw,dx,e,fu,fv,fw,fx,olde,tol1,tol2,&
    u,u1,u2,v,w,x,xm
```

Comments following will point out only differences from the routine `brent`. Read that routine first.

```
LOGICAL :: ok1,ok2
```

Will be used as flags for whether proposed steps are acceptable or not.

```
a=min(ax,cx)
b=max(ax,cx)
v=bx
w=v
x=v
e=0.0
fx=func(x)
fv=fx
fw=fx
dx=dfunc(x)
```

All our housekeeping chores are doubled by the necessity of moving derivative values around as well as function values.

```
dv=dx
dw=dx
do iter=1,ITMAX
    xm=0.5_sp*(a+b)
    tol1=tol*abs(x)+ZEPS
    tol2=2.0_sp*tol1
    if (abs(x-xm) <= (tol2-0.5_sp*(b-a))) exit
    if (abs(e) > tol1) then
        d1=2.0_sp*(b-a)
```

Initialize these `d`'s to an out-of-bracket value.

```
        d2=d1
        if (dw /= dx) d1=(w-x)*dx/(dx-dw)
```

Secant method with each point.

```
        if (dv /= dx) d2=(v-x)*dx/(dx-dv)
```

Which of these two estimates of `d` shall we take? We will insist that they be within the bracket, and on the side pointed to by the derivative at `x`:

```
        u1=x+d1
        u2=x+d2
        ok1=((a-u1)*(u1-b) > 0.0) .and. (dx*d1 <= 0.0)
        ok2=((a-u2)*(u2-b) > 0.0) .and. (dx*d2 <= 0.0)
        olde=e
```

Movement on the step before last.

```
        e=d
        if (ok1 .or. ok2) then
```

Take only an acceptable `d`, and if both are acceptable, then take the smallest one.

```
            if (ok1 .and. ok2) then
                d=merge(d1,d2, abs(d1) < abs(d2))
            else
                d=merge(d1,d2,ok1)
            end if
            if (abs(d) <= abs(0.5_sp*olde)) then
                u=x+d
                if (u-a < tol2 .or. b-u < tol2) &
                    d=sign(tol1,xm-x)
            else
```

```
                e=merge(a,b, dx >= 0.0)-x
                  Decide which segment by the sign of the derivative.
                d=0.5_sp*e                          Bisect, not golden section.
            end if
        else
            e=merge(a,b, dx >= 0.0)-x
            d=0.5_sp*e                              Bisect, not golden section.
        end if
    else
        e=merge(a,b, dx >= 0.0)-x
        d=0.5_sp*e                                  Bisect, not golden section.
    end if
    if (abs(d) >= tol1) then
        u=x+d
        fu=func(u)
    else
        u=x+sign(tol1,d)
        fu=func(u)                                  If the minimum step in the downhill
        if (fu > fx) exit                               direction takes us uphill, then we
    end if                                              are done.
    du=dfunc(u)                                     Now all the housekeeping, sigh.
    if (fu <= fx) then
        if (u >= x) then
            a=x
        else
            b=x
        end if
        call mov3(v,fv,dv,w,fw,dw)
        call mov3(w,fw,dw,x,fx,dx)
        call mov3(x,fx,dx,u,fu,du)
    else
        if (u < x) then
            a=u
        else
            b=u
        end if
        if (fu <= fw .or. w == x) then
            call mov3(v,fv,dv,w,fw,dw)
            call mov3(w,fw,dw,u,fu,du)
        else if (fu <= fv .or. v == x .or. v == w) then
            call mov3(v,fv,dv,u,fu,du)
        end if
    end if
end do
if (iter > ITMAX) call nrerror('dbrent: exceeded maximum iterations')
xmin=x
dbrent=fx
CONTAINS
SUBROUTINE mov3(a,b,c,d,e,f)
REAL(SP), INTENT(IN) :: d,e,f
REAL(SP), INTENT(OUT) :: a,b,c
a=d
b=e
c=f
END SUBROUTINE mov3
END FUNCTION dbrent
```

⋆ ⋆ ⋆

```
SUBROUTINE amoeba(p,y,ftol,func,iter)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,iminloc,nrerror,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=5000
REAL(SP), PARAMETER :: TINY=1.0e-10
```

Minimization of the function `func` in N dimensions by the downhill simplex method of Nelder and Mead. The $(N+1) \times N$ matrix `p` is input. Its $N+1$ rows are N-dimensional vectors that are the vertices of the starting simplex. Also input is the vector `y` of length $N+1$, whose components must be preinitialized to the values of `func` evaluated at the $N+1$ vertices (rows) of `p`; and `ftol` the fractional convergence tolerance to be achieved in the function value (n.b.!). On output, `p` and `y` will have been reset to $N+1$ new points all within `ftol` of a minimum function value, and `iter` gives the number of function evaluations taken.

Parameters: The maximum allowed number of function evaluations, and a small number.

```
INTEGER(I4B) :: ihi,ndim                          Global variables.
REAL(SP), DIMENSION(size(p,2)) :: psum
call amoeba_private
CONTAINS

SUBROUTINE amoeba_private
IMPLICIT NONE
INTEGER(I4B) :: i,ilo,inhi
REAL(SP) :: rtol,ysave,ytry,ytmp
ndim=assert_eq(size(p,2),size(p,1)-1,size(y)-1,'amoeba')
iter=0
psum(:)=sum(p(:,:),dim=1)
do                                                Iteration loop.
    ilo=iminloc(y(:))                             Determine which point is the highest (worst),
    ihi=imaxloc(y(:))                                 next-highest, and lowest (best).
    ytmp=y(ihi)
    y(ihi)=y(ilo)
    inhi=imaxloc(y(:))
    y(ihi)=ytmp
    rtol=2.0_sp*abs(y(ihi)-y(ilo))/(abs(y(ihi))+abs(y(ilo))+TINY)
      Compute the fractional range from highest to lowest and return if satisfactory.
    if (rtol < ftol) then                         If returning, put best point and value in slot
        call swap(y(1),y(ilo))                        1.
        call swap(p(1,:),p(ilo,:))
        RETURN
    end if
    if (iter >= ITMAX) call nrerror('ITMAX exceeded in amoeba')
      Begin a new iteration. First extrapolate by a factor −1 through the face of the simplex
      across from the high point, i.e., reflect the simplex from the high point.
    ytry=amotry(-1.0_sp)
    iter=iter+1
    if (ytry <= y(ilo)) then                      Gives a result better than the best point, so
        ytry=amotry(2.0_sp)                           try an additional extrapolation by a fac-
        iter=iter+1                                   tor of 2.
    else if (ytry >= y(inhi)) then                The reflected point is worse than the sec-
        ysave=y(ihi)                                  ond highest, so look for an intermediate
        ytry=amotry(0.5_sp)                           lower point, i.e., do a one-dimensional
        iter=iter+1                                   contraction.
```

```
        if (ytry >= ysave) then
              Can't seem to get rid of that high point. Better contract around the lowest
              (best) point.
            p(:,:)=0.5_sp*(p(:,:)+spread(p(ilo,:),1,size(p,1)))
            do i=1,ndim+1
                if (i /= ilo) y(i)=func(p(i,:))
            end do
            iter=iter+ndim                    Keep track of function evaluations.
            psum(:)=sum(p(:,:),dim=1)
        end if
    end if
end do                                        Go back for the test of doneness and the next
END SUBROUTINE amoeba_private                     iteration.

FUNCTION amotry(fac)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fac
REAL(SP) :: amotry
   Extrapolates by a factor fac through the face of the simplex across from the high point,
   tries it, and replaces the high point if the new point is better.
REAL(SP) :: fac1,fac2,ytry
REAL(SP), DIMENSION(size(p,2)) :: ptry
fac1=(1.0_sp-fac)/ndim
fac2=fac1-fac
ptry(:)=psum(:)*fac1-p(ihi,:)*fac2
ytry=func(ptry)                               Evaluate the function at the trial point.
if (ytry < y(ihi)) then                       If it's better than the highest, then replace
    y(ihi)=ytry                                   the highest.
    psum(:)=psum(:)-p(ihi,:)+ptry(:)
    p(ihi,:)=ptry(:)
end if
amotry=ytry
END FUNCTION amotry
END SUBROUTINE amoeba
```

f90 The only action taken by the subroutine `amoeba` is to call the internal subroutine `amoeba_private`. Why this structure? The reason has to do with meeting the twin goals of data hiding (especially for "safe" scope of variables) and program readability. The situation is this: Logically, `amoeba` does most of the calculating, but calls an internal subroutine `amotry` at several different points, with several values of the parameter `fac`. However, `fac` is not the only piece of data that must be shared with `amotry`; the latter also needs access to several shared variables (`ihi`, `ndim`, `psum`) and arguments of `amoeba` (`p`, `y`, `func`).

The obvious (but not best) way of coding this would be to put the computational guts in `amoeba`, with `amotry` as the sole internal subprogram. Assuming that `fac` is passed as an argument to `amotry` (it being the parameter that is being rapidly altered), one must decide whether to pass all the other quantities to `amotry` (i) as additional arguments (as is done in the Fortran 77 version), or (ii) "automatically," i.e., doing nothing except using the fact that an internal subprogram has automatic access to all of its host's entities. Each of these choices has strong disadvantages. Choice (i) is inefficient (all those arguments) and also obscures the fact that `fac` is the primary changing argument. Choice (ii) makes the program extremely difficult to read, because it wouldn't be obvious without careful cross-comparison of the routines *which* variables in `amoeba` are actually global variables that are used by `amotry`.

Choice (ii) is also "unsafe scoping" because it gives a nontrivially complicated internal subprogram, `amotry`, access to all the variables in its host. A common and difficult-to-find bug is the accidental alteration of a variable that one "thought"

was local, but is actually shared. (Simple variables like `i`, `j`, and `n` are the most common culprits.)

We are therefore led to reject both choice (i) and choice (ii) in favor of a structure previously described in the subsection on Scope, Visibility, and Data Hiding in §21.5. The guts of `amoeba` are put in `amoeba_private`, a *sister routine* to `amotry`. These two siblings have mutually private name spaces. However, any variables that they need to share (including the top-level arguments of `amoeba`) are declared as variables in the enclosing `amoeba` routine. The presence of these "global variables" serves as a warning flag to the reader that data are shared between routines.

An alternative attractive way of coding the above situation would be to use a module containing `amoeba` and `amotry`. Everything would be declared private except the name `amoeba`. The global variables would be at the top level, and the arguments of `amoeba` that need to be passed to `amotry` would be handled by pointers among the global variables. Unfortunately, Fortran 90 does not support pointers to functions. Sigh!

`ilo=iminloc...ihi=imaxloc...` See discussion of these functions on p. 1017.

`call swap(y(1)...call swap(p(1,:)...` Here the `swap` routine in `nrutil` is called once with a scalar argument and once with a vector argument. Inside `nrutil` scalar and vector versions have been overloaded onto the single name `swap`, hiding all the implementation details from the calling routine.

⋆ ⋆ ⋆

```
SUBROUTINE powell(p,xi,ftol,iter,fret)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : linmin
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: xi
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: fret
INTERFACE
    FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: TINY=1.0e-25_sp
```

Minimization of a function `func` of N variables. (`func` is not an argument, it is a fixed function name.) Input consists of an initial starting point `p`, a vector of length N; an initial $N \times N$ matrix `xi` whose columns contain the initial set of directions (usually the N unit vectors); and `ftol`, the fractional tolerance in the function value such that failure to decrease by more than this amount on one iteration signals doneness. On output, `p` is set to the best point found, `xi` is the then-current direction set, `fret` is the returned function value at `p`, and `iter` is the number of iterations taken. The routine `linmin` is used.

Parameters: Maximum allowed iterations, and a small number.

```
INTEGER(I4B) :: i,ibig,n
REAL(SP) :: del,fp,fptt,t
REAL(SP), DIMENSION(size(p)) :: pt,ptt,xit
n=assert_eq(size(p),size(xi,1),size(xi,2),'powell')
fret=func(p)
```

```
pt(:)=p(:)                                  Save the initial point.
iter=0
do
    iter=iter+1
    fp=fret
    ibig=0
    del=0.0                                 Will be the biggest function decrease.
    do i=1,n                                Loop over all directions in the set.
        xit(:)=xi(:,i)                      Copy the direction,
        fptt=fret
        call linmin(p,xit,fret)             minimize along it,
        if (fptt-fret > del) then           and record it if it is the largest decrease so
            del=fptt-fret                       far.
            ibig=i
        end if
    end do
    if (2.0_sp*(fp-fret) <= ftol*(abs(fp)+abs(fret))+TINY) RETURN
      Termination criterion.
    if (iter == ITMAX) call &
        nrerror('powell exceeding maximum iterations')
    ptt(:)=2.0_sp*p(:)-pt(:)                Construct the extrapolated point and the av-
    xit(:)=p(:)-pt(:)                           erage direction moved. Save the old start-
    pt(:)=p(:)                                  ing point.
    fptt=func(ptt)                          Function value at extrapolated point.
    if (fptt >= fp) cycle                   One reason not to use new direction.
    t=2.0_sp*(fp-2.0_sp*fret+fptt)*(fp-fret-del)**2-del*(fp-fptt)**2
    if (t >= 0.0) cycle                     Other reason not to use new direction.
    call linmin(p,xit,fret)                 Move to minimum of the new direction,
    xi(:,ibig)=xi(:,n)                      and save the new direction.
    xi(:,n)=xit(:)
end do                                      Back for another iteration.
END SUBROUTINE powell
```

⋆ ⋆ ⋆

```
MODULE f1dim_mod                Used for communication from linmin to f1dim.
USE nrtype
INTEGER(I4B) :: ncom
REAL(SP), DIMENSION(:), POINTER :: pcom,xicom
CONTAINS

FUNCTION f1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: f1dim
   Used by linmin as the one-dimensional function passed to mnbrak and brent.
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:), ALLOCATABLE :: xt
allocate(xt(ncom))
xt(:)=pcom(:)+x*xicom(:)
f1dim=func(xt)
deallocate(xt)
END FUNCTION f1dim
END MODULE f1dim_mod
```

```
SUBROUTINE linmin(p,xi,fret)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : mnbrak,brent
USE f1dim_mod
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), TARGET, INTENT(INOUT) :: p,xi
REAL(SP), PARAMETER :: TOL=1.0e-4_sp
```

Given an N-dimensional point p and an N-dimensional direction xi, both vectors of length N, moves and resets p to where the fixed-name function func takes on a minimum along the direction xi from p, and replaces xi by the actual vector displacement that p was moved. Also returns as fret the value of func at the returned location p. This is actually all accomplished by calling the routines mnbrak and brent.

Parameter: Tolerance passed to brent.

```
REAL(SP) :: ax,bx,fa,fb,fx,xmin,xx
ncom=assert_eq(size(p),size(xi),'linmin')
pcom=>p                          Communicate the global variables to f1dim.
xicom=>xi
ax=0.0                           Initial guess for brackets.
xx=1.0
call mnbrak(ax,xx,bx,fa,fx,fb,f1dim)
fret=brent(ax,xx,bx,f1dim,TOL,xmin)
xi=xmin*xi                       Construct the vector results to return.
p=p+xi
END SUBROUTINE linmin
```

f90 USE f1dim_mod At first sight this situation is like the one involving USE fminln in newt on p. 1197: We want to pass arrays p and xi from linmin to f1dim without having them be arguments of f1dim. If you recall the discussion in §21.5 and on p. 1197, there are two ways of effecting this: via pointers or via allocatable arrays. There is an important difference here, however. The arrays p and xi are themselves arguments of linmin, and so cannot be allocatable arrays in the module. If we did want to use allocatable arrays in the module, we would have to copy p and xi into them. The pointer implementation is much more elegant, since no unnecessary copying is required. The construction here is identical to the one in fminln and newt, except that p and xi are arguments instead of automatic arrays.

⋆ ⋆ ⋆

MODULE df1dim_mod Used for communication from dlinmin to f1dim and df1dim.

```
USE nrtype
INTEGER(I4B) :: ncom
REAL(SP), DIMENSION(:), POINTER :: pcom,xicom
CONTAINS
FUNCTION f1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: f1dim
```

Used by dlinmin as the one-dimensional function passed to mnbrak.

```
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
REAL(SP), DIMENSION(:), ALLOCATABLE :: xt
```

```
allocate(xt(ncom))
xt(:)=pcom(:)+x*xicom(:)
f1dim=func(xt)
deallocate(xt)
END FUNCTION f1dim
```

```
FUNCTION df1dim(x)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP) :: df1dim
```

Used by dlinmin as the one-dimensional function passed to dbrent.

```
INTERFACE
    FUNCTION dfunc(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: dfunc
    END FUNCTION dfunc
END INTERFACE
REAL(SP), DIMENSION(:), ALLOCATABLE :: xt,df
allocate(xt(ncom),df(ncom))
xt(:)=pcom(:)+x*xicom(:)
df(:)=dfunc(xt)
df1dim=dot_product(df,xicom)
deallocate(xt,df)
END FUNCTION df1dim
END MODULE df1dim_mod
```

```
SUBROUTINE dlinmin(p,xi,fret)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : mnbrak,dbrent
USE df1dim_mod
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), TARGET :: p,xi
REAL(SP), PARAMETER :: TOL=1.0e-4_sp
```

Given an N-dimensional point p and an N-dimensional direction xi, both vectors of length N, moves and resets p to where the fixed-name function func takes on a minimum along the direction xi from p, and replaces xi by the actual vector displacement that p was moved. Also returns as fret the value of func at the returned location p. This is actually all accomplished by calling the routines mnbrak and dbrent. dfunc is a fixed-name user-supplied function that computes the gradient of func.

Parameter: Tolerance passed to dbrent.

```
REAL(SP) :: ax,bx,fa,fb,fx,xmin,xx
ncom=assert_eq(size(p),size(xi),'dlinmin')
pcom=>p                          Communicate the global variables to f1dim.
xicom=>xi
ax=0.0                           Initial guess for brackets.
xx=1.0
call mnbrak(ax,xx,bx,fa,fx,fb,f1dim)
fret=dbrent(ax,xx,bx,f1dim,df1dim,TOL,xmin)
xi=xmin*xi                       Construct the vector results to return.
p=p+xi
END SUBROUTINE dlinmin
```

f90 USE df1dim_mod See discussion of USE f1dim_mod on p. 1212.

⋆ ⋆ ⋆

```
SUBROUTINE frprmn(p,ftol,iter,fret)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : linmin
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: ftol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP) :: func
    END FUNCTION func

    FUNCTION dfunc(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP), DIMENSION(size(p)) :: dfunc
    END FUNCTION dfunc
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: EPS=1.0e-10_sp
```

Given a starting point `p` that is a vector of length N, Fletcher-Reeves-Polak-Ribiere minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence tolerance on the function value is input as `ftol`. Returned quantities are `p` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `linmin` is called to perform line minimizations.

Parameters: `ITMAX` is the maximum allowed number of iterations; `EPS` is a small number to rectify the special case of converging to exactly zero function value.

```
INTEGER(I4B) :: its
REAL(SP) :: dgg,fp,gam,gg
REAL(SP), DIMENSION(size(p)) :: g,h,xi
fp=func(p)                          Initializations.
xi=dfunc(p)
g=-xi
h=g
xi=h
do its=1,ITMAX                      Loop over iterations.
    iter=its
    call linmin(p,xi,fret)          Next statement is the normal return:
    if (2.0_sp*abs(fret-fp) <= ftol*(abs(fret)+abs(fp)+EPS)) RETURN
    fp=fret
    xi=dfunc(p)
    gg=dot_product(g,g)
!   dgg=dot_product(xi,xi)          This statement for Fletcher-Reeves.
    dgg=dot_product(xi+g,xi)        This statement for Polak-Ribiere.
    if (gg == 0.0) RETURN           Unlikely. If gradient is exactly zero then we are al-
    gam=dgg/gg                          ready done.
    g=-xi
    h=g+gam*h
    xi=h
end do
call nrerror('frprmn: maximum iterations exceeded')
END SUBROUTINE frprmn
```

★ ★ ★

```
SUBROUTINE dfpmin(p,gtol,iter,fret,func,dfunc)
USE nrtype; USE nrutil, ONLY : nrerror,outerprod,unit_matrix,vabs
USE nr, ONLY : lnsrch
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: iter
REAL(SP), INTENT(IN) :: gtol
REAL(SP), INTENT(OUT) :: fret
REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP) :: func
    END FUNCTION func

    FUNCTION dfunc(p)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: p
    REAL(SP), DIMENSION(size(p)) :: dfunc
    END FUNCTION dfunc
END INTERFACE
INTEGER(I4B), PARAMETER :: ITMAX=200
REAL(SP), PARAMETER :: STPMX=100.0_sp,EPS=epsilon(p),TOLX=4.0_sp*EPS
```

Given a starting point p that is a vector of length N, the Broyden-Fletcher-Goldfarb-Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence requirement on zeroing the gradient is input as `gtol`. Returned quantities are p (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `lnsrch` is called to perform approximate line minimizations.

Parameters: `ITMAX` is the maximum allowed number of iterations; `STPMX` is the scaled maximum step length allowed in line searches; `EPS` is the machine precision; `TOLX` is the convergence criterion on x values.

```
INTEGER(I4B) :: its
LOGICAL :: check
REAL(SP) :: den,fac,fad,fae,fp,stpmax,sumdg,sumxi
REAL(SP), DIMENSION(size(p)) :: dg,g,hdg,pnew,xi
REAL(SP), DIMENSION(size(p),size(p)) :: hessin
fp=func(p)                                  Calculate starting function value and gradi-
g=dfunc(p)                                      ent.
call unit_matrix(hessin)                    Initialize inverse Hessian to the unit matrix.
xi=-g                                       Initial line direction.
stpmax=STPMX*max(vabs(p),real(size(p),sp))
do its=1,ITMAX                              Main loop over the iterations.
    iter=its
    call lnsrch(p,fp,g,xi,pnew,fret,stpmax,check,func)
      The new function evaluation occurs in lnsrch; save the function value in fp for the next
      line search. It is usually safe to ignore the value of check.
    fp=fret
    xi=pnew-p                               Update the line direction,
    p=pnew                                  and the current point.
    if (maxval(abs(xi)/max(abs(p),1.0_sp)) < TOLX) RETURN
      Test for convergence on \Delta x.
    dg=g                                    Save the old gradient,
    g=dfunc(p)                              and get the new gradient.
    den=max(fret,1.0_sp)
    if (maxval(abs(g)*max(abs(p),1.0_sp)/den) < gtol) RETURN
      Test for convergence on zero gradient.
    dg=g-dg                                 Compute difference of gradients,
    hdg=matmul(hessin,dg)                   and difference times current matrix.
    fac=dot_product(dg,xi)                  Calculate dot products for the denominators.
    fae=dot_product(dg,hdg)
    sumdg=dot_product(dg,dg)
```

```
    sumxi=dot_product(xi,xi)
    if (fac > sqrt(EPS*sumdg*sumxi)) then        Skip update if fac not sufficiently
        fac=1.0_sp/fac                               positive.
        fad=1.0_sp/fae
        dg=fac*xi-fad*hdg                    Vector that makes BFGS different from DFP.
        hessin=hessin+fac*outerprod(xi,xi)-&     The BFGS updating formula.
            fad*outerprod(hdg,hdg)+fae*outerprod(dg,dg)
    end if
    xi=-matmul(hessin,g)                     Now calculate the next direction to go,
end do                                       and go back for another iteration.
call nrerror('dfpmin: too many iterations')
END SUBROUTINE dfpmin
```

f90 `call unit_matrix(hessin)` The `unit_matrix` routine in `nrutil` does exactly what its name suggests. The routine `dfpmin` makes use of `outerprod` from `nrutil`, as well as the matrix intrinsics `matmul` and `dot_product`, to simplify and parallelize the coding.

⋆ ⋆ ⋆

```
SUBROUTINE simplx(a,m1,m2,m3,icase,izrov,iposv)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,ifirstloc,imaxloc,&
    nrerror,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: m1,m2,m3
INTEGER(I4B), INTENT(OUT) :: icase
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: izrov,iposv
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Simplex method for linear programming. Input parameters `a`, `m1`, `m2`, and `m3`, and output parameters `a`, `icase`, `izrov`, and `iposv` are described above the routine in Vol. 1. Dimensions are $(M+2) \times (N+1)$ for `a`, M for `iposv`, N for `izrov`, with `m1` + `m2` + `m3` = M. Parameter: `EPS` is the absolute precision, which should be adjusted to the scale of your variables.

```
INTEGER(I4B) :: ip,k,kh,kp,nl1,m,n
INTEGER(I4B), DIMENSION(size(a,2)) :: l1
INTEGER(I4B), DIMENSION(m2) :: l3
REAL(SP) :: bmax
LOGICAL(LGT) :: init
m=assert_eq(size(a,1)-2,size(iposv),'simplx: m')
n=assert_eq(size(a,2)-1,size(izrov),'simplx: n')
if (m /= m1+m2+m3) call nrerror('simplx: bad input constraint counts')
if (any(a(2:m+1,1) < 0.0)) call nrerror('bad input tableau in simplx')
```

Constants b_i must be nonnegative.

```
nl1=n
l1(1:n)=arth(1,1,n)
```

Initialize index list of columns admissible for exchange.

```
izrov(:)=l1(1:n)                             Initially make all variables right-hand.
iposv(:)=n+arth(1,1,m)
```

Initial left-hand variables. `m1` type constraints are represented by having their slack variable initially left-hand, with no artificial variable. `m2` type constraints have their slack variable initially left-hand, with a minus sign, and their artificial variable handled implicitly during their first exchange. `m3` type constraints have their artificial variable initially left-hand.

```
init=.true.
phase1: do
    if (init) then                           Initial pass only.
        if (m2+m3 == 0) exit phase1          Origin is a feasible solution. Go to phase two.
        init=.false.
        l3(1:m2)=1
```

Initialize list of `m2` constraints whose slack variables have never been exchanged out of the initial basis.

```
        a(m+2,1:n+1)=-sum(a(m1+2:m+1,1:n+1),dim=1)      Compute the auxiliary objec-
    end if                                                   tive function.
```

```
    if (nl1 > 0) then
        kp=l1(imaxloc(a(m+2,l1(1:nl1)+1)))        Find the maximum coefficient of the
        bmax=a(m+2,kp+1)                              auxiliary objective function.
    else
        bmax=0.0
    end if
    phase1a: do
        if (bmax <= EPS .and. a(m+2,1) < -EPS) then
              Auxiliary objective function is still negative and can't be improved, hence no
              feasible solution exists.
            icase=-1
            RETURN
        else if (bmax <= EPS .and. a(m+2,1) <= EPS) then
              Auxiliary objective function is zero and can't be improved. This signals that we
              have a feasible starting vector. Clean out the artificial variables corresponding
              to any remaining equality constraints and then eventually exit phase one.
            do ip=m1+m2+1,m
                if (iposv(ip) == ip+n) then       Found an artificial variable for an equal-
                    if (nl1 > 0) then                 ity constraint.
                        kp=l1(imaxloc(abs(a(ip+1,l1(1:nl1)+1))))
                        bmax=a(ip+1,kp+1)
                    else
                        bmax=0.0
                    end if
                    if (bmax > EPS) exit phase1a      Exchange with column correspond-
                end if                                    ing to maximum pivot ele-
            end do                                        ment in row.
            where (spread(l3(1:m2),2,n+1) == 1) &
                a(m1+2:m1+m2+1,1:n+1)=-a(m1+2:m1+m2+1,1:n+1)
                Change sign of row for any m2 constraints still present from the initial basis.
            exit phase1                           Go to phase two.
        end if
        call simp1                                Locate a pivot element (phase one).
        if (ip == 0) then                         Maximum of auxiliary objective function is
            icase=-1                                  unbounded, so no feasible solution ex-
            RETURN                                    ists.
        end if
        exit phase1a
    end do phase1a
    call simp2(m+1,n)                             Exchange a left- and a right-hand variable.
    if (iposv(ip) >= n+m1+m2+1) then              Exchanged out an artificial variable for an
        k=ifirstloc(l1(1:nl1) == kp)                  equality constraint. Make sure it stays
        nl1=nl1-1                                     out by removing it from the l1 list.
        l1(k:nl1)=l1(k+1:nl1+1)
    else
        kh=iposv(ip)-m1-n
        if (kh >= 1) then                         Exchanged out an m2 type constraint.
            if (l3(kh) /= 0) then                 If it's the first time, correct the pivot col-
                l3(kh)=0                              umn for the minus sign and the implicit
                a(m+2,kp+1)=a(m+2,kp+1)+1.0_sp        artificial variable.
                a(1:m+2,kp+1)=-a(1:m+2,kp+1)
            end if
        end if
    end if
    call swap(izrov(kp),iposv(ip))                Update lists of left- and right-hand variables.
end do phase1                                     If still in phase one, go back again.
phase2: do
      We have an initial feasible solution. Now optimize it.
    if (nl1 > 0) then
        kp=l1(imaxloc(a(1,l1(1:nl1)+1)))          Test the z-row for doneness.
        bmax=a(1,kp+1)
    else
        bmax=0.0
    end if
```

```
    if (bmax <= EPS) then                    Done. Solution found. Return with the good
        icase=0                                  news.
        RETURN
    end if
    call simp1                               Locate a pivot element (phase two).
    if (ip == 0) then                        Objective function is unbounded. Report and
        icase=1                                  return.
        RETURN
    end if
    call simp2(m,n)                          Exchange a left- and a right-hand variable,
    call swap(izrov(kp),iposv(ip))           update lists of left- and right-hand variables,
end do phase2                                and return for another iteration.
CONTAINS

SUBROUTINE simp1
    Locate a pivot element, taking degeneracy into account.
IMPLICIT NONE
INTEGER(I4B) :: i,k
REAL(SP) :: q,q0,q1,qp
ip=0
i=ifirstloc(a(2:m+1,kp+1) < -EPS)
if (i > m) RETURN                            No possible pivots. Return with message.
q1=-a(i+1,1)/a(i+1,kp+1)
ip=i
do i=ip+1,m
    if (a(i+1,kp+1) < -EPS) then
        q=-a(i+1,1)/a(i+1,kp+1)
        if (q < q1) then
            ip=i
            q1=q
        else if (q == q1) then               We have a degeneracy.
            do k=1,n
                qp=-a(ip+1,k+1)/a(ip+1,kp+1)
                q0=-a(i+1,k+1)/a(i+1,kp+1)
                if (q0 /= qp) exit
            end do
            if (q0 < qp) ip=i
        end if
    end if
end do
END SUBROUTINE simp1

SUBROUTINE simp2(i1,k1)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: i1,k1
    Matrix operations to exchange a left-hand and right-hand variable (see text).
INTEGER(I4B) :: ip1,kp1
REAL(SP) :: piv
INTEGER(I4B), DIMENSION(k1) :: icol
INTEGER(I4B), DIMENSION(i1) :: irow
INTEGER(I4B), DIMENSION(max(i1,k1)+1) :: itmp
ip1=ip+1
kp1=kp+1
piv=1.0_sp/a(ip1,kp1)
itmp(1:k1+1)=arth(1,1,k1+1)
icol=pack(itmp(1:k1+1),itmp(1:k1+1) /= kp1)
itmp(1:i1+1)=arth(1,1,i1+1)
irow=pack(itmp(1:i1+1),itmp(1:i1+1) /= ip1)
a(irow,kp1)=a(irow,kp1)*piv
a(irow,icol)=a(irow,icol)-outerprod(a(irow,kp1),a(ip1,icol))
a(ip1,icol)=-a(ip1,icol)*piv
a(ip1,kp1)=piv
END SUBROUTINE simp2
END SUBROUTINE simplx
```

`main_procedure: do` The routine `simplx` makes extensive use of named do-loops to control the program flow. The various `exit` statements have the names of the do-loops attached to them so we can easily tell where control is being transferred to. We believe that it is almost never necessary to use `goto` statements: Code will always be clearer with well-constructed block structures.

`phase1a: do...end do phase1a` This is not a real do-loop: It is executed only once, as you can see from the unconditional `exit` before the `end do`. We use this construction to define a block of code that is traversed once but that has several possible exit points.

```
where (spread(l3(1:m12-m1),2,n+1) == 1) &
   a(m1+2:m12+1,1:n+1)=-a(m1+2:m12+1,1:n+1)
```

These lines are equivalent to

```
do i=m1+1,m12
   if (l3(i-m1) == 1) a(i+1,1:n+1)=-a(i+1,1:n+1)
end do
```

⋆ ⋆ ⋆

```
SUBROUTINE anneal(x,y,iorder)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,swap
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
```

This algorithm finds the shortest round-trip path to N cities whose coordinates are in the length N arrays `x`, `y`. The length N array `iorder` specifies the order in which the cities are visited. On input, the elements of `iorder` may be set to any permutation of the numbers $1 \ldots N$. This routine will return the best alternative path it can find.

```
INTEGER(I4B), DIMENSION(6) :: n
INTEGER(I4B) :: i1,i2,j,k,nlimit,ncity,nn,nover,nsucc
REAL(SP) :: de,harvest,path,t,tfactr
LOGICAL(LGT) :: ans
ncity=assert_eq(size(x),size(y),size(iorder),'anneal')
nover=100*ncity                    Maximum number of paths tried at any temperature,
nlimit=10*ncity                    and of successful path changes before continuing.
tfactr=0.9_sp                      Annealing schedule: t is reduced by this factor on
t=0.5_sp                               each step.
path=sum(alen_v(x(iorder(1:ncity-1)),x(iorder(2:ncity)),&
    y(iorder(1:ncity-1)),y(iorder(2:ncity))))      Calculate initial path length.
i1=iorder(ncity)                   Close the loop by tying path ends to-
i2=iorder(1)                           gether.
path=path+alen(x(i1),x(i2),y(i1),y(i2))
do j=1,100                         Try up to 100 temperature steps.
    nsucc=0
    do k=1,nover
        do
            call ran1(harvest)
            n(1)=1+int(ncity*harvest)          Choose beginning of segment ...
            call ran1(harvest)
            n(2)=1+int((ncity-1)*harvest)      ... and end of segment.
            if (n(2) >= n(1)) n(2)=n(2)+1
            nn=1+mod((n(1)-n(2)+ncity-1),ncity)    nn is the number of cities not on
            if (nn >= 3) exit                          the segment.
        end do
```

```
        call ran1(harvest)
          Decide whether to do a reversal or a transport.
        if (harvest < 0.5_sp) then              Do a transport.
            call ran1(harvest)
            n(3)=n(2)+int(abs(nn-2)*harvest)+1
            n(3)=1+mod(n(3)-1,ncity)            Transport to a location not on the path.
            call trncst(x,y,iorder,n,de)        Calculate cost.
            call metrop(de,t,ans)               Consult the oracle.
            if (ans) then
                nsucc=nsucc+1
                path=path+de
                call trnspt(iorder,n)           Carry out the transport.
            end if
        else                                    Do a path reversal.
            call revcst(x,y,iorder,n,de)        Calculate cost.
            call metrop(de,t,ans)               Consult the oracle.
            if (ans) then
                nsucc=nsucc+1
                path=path+de
                call revers(iorder,n)           Carry out the reversal.
            end if
        end if
        if (nsucc >= nlimit) exit               Finish early if we have enough successful
                                                    changes.
    end do
    write(*,*)
    write(*,*) 'T =',t,' Path Length =',path
    write(*,*) 'Successful Moves: ',nsucc
    t=t*tfactr                                  Annealing schedule.
    if (nsucc == 0) RETURN                      If no success, we are done.
end do
CONTAINS

FUNCTION alen(x1,x2,y1,y2)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1,x2,y1,y2
REAL(SP) :: alen
    Computes distance between two cities.
alen=sqrt((x2-x1)**2+(y2-y1)**2)
END FUNCTION alen

FUNCTION alen_v(x1,x2,y1,y2)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,x2,y1,y2
REAL(SP), DIMENSION(size(x1)) :: alen_v
    Computes distances between pairs of cities.
alen_v=sqrt((x2-x1)**2+(y2-y1)**2)
END FUNCTION alen_v

SUBROUTINE metrop(de,t,ans)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: de,t
LOGICAL(LGT), INTENT(OUT) :: ans
    Metropolis algorithm. ans is a logical variable that issues a verdict on whether to accept a
    reconfiguration that leads to a change de in the objective function. If de<0, ans=.true.,
    while if de>0, ans is only .true. with probability exp(-de/t), where t is a temperature
    determined by the annealing schedule.
call ran1(harvest)
ans=(de < 0.0) .or. (harvest < exp(-de/t))
END SUBROUTINE metrop

SUBROUTINE revcst(x,y,iorder,n,de)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: n
REAL(SP), INTENT(OUT) :: de
```

This subroutine returns the value of the cost function for a proposed path reversal. The arrays x and y give the coordinates of these cities. iorder holds the present itinerary. The first two values n(1) and n(2) of array n give the starting and ending cities along the path segment which is to be reversed. On output, de is the cost of making the reversal. The actual reversal is not performed by this routine.

```
INTEGER(I4B) :: ncity
REAL(SP), DIMENSION(4) :: xx,yy
ncity=size(x)
n(3)=1+mod((n(1)+ncity-2),ncity)          Find the city before n(1) ...
n(4)=1+mod(n(2),ncity)                    ... and the city after n(2).
xx(1:4)=x(iorder(n(1:4)))                 Find coordinates for the four cities involved.
yy(1:4)=y(iorder(n(1:4)))
de=-alen(xx(1),xx(3),yy(1),yy(3))&        Calculate cost of disconnecting the segment
    -alen(xx(2),xx(4),yy(2),yy(4))&           at both ends and reconnecting in the op-
    +alen(xx(1),xx(4),yy(1),yy(4))&           posite order.
    +alen(xx(2),xx(3),yy(2),yy(3))
END SUBROUTINE revcst
```

```
SUBROUTINE revers(iorder,n)
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
```

This routine performs a path segment reversal. iorder is an input array giving the present itinerary. The vector n has as its first four elements the first and last cities n(1), n(2) of the path segment to be reversed, and the two cities n(3) and n(4) that immediately precede and follow this segment. n(3) and n(4) are found by subroutine revcst. On output, iorder contains the segment from n(1) to n(2) in reversed order.

```
INTEGER(I4B) :: j,k,l,nn,ncity
ncity=size(iorder)
nn=(1+mod(n(2)-n(1)+ncity,ncity))/2       This many cities must be swapped to effect
do j=1,nn                                     the reversal.
    k=1+mod((n(1)+j-2),ncity)             Start at the ends of the segment and swap
    l=1+mod((n(2)-j+ncity),ncity)             pairs of cities, moving toward the cen-
    call swap(iorder(k),iorder(l))            ter.
end do
END SUBROUTINE revers
```

```
SUBROUTINE trncst(x,y,iorder,n,de)
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: n
REAL(SP), INTENT(OUT) :: de
```

This subroutine returns the value of the cost function for a proposed path segment transport. Arrays x and y give the city coordinates. iorder is an array giving the present itinerary. The first three elements of array n give the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, de is the cost of the change. The actual transport is not performed by this routine.

```
INTEGER(I4B) :: ncity
REAL(SP), DIMENSION(6) :: xx,yy
ncity=size(x)
n(4)=1+mod(n(3),ncity)                    Find the city following n(3) ...
n(5)=1+mod((n(1)+ncity-2),ncity)          ... and the one preceding n(1) ...
n(6)=1+mod(n(2),ncity)                    ... and the one following n(2).
xx(1:6)=x(iorder(n(1:6)))                 Determine coordinates for the six cities in-
yy(1:6)=y(iorder(n(1:6)))                     volved.
de=-alen(xx(2),xx(6),yy(2),yy(6))&        Calculate the cost of disconnecting the path
    -alen(xx(1),xx(5),yy(1),yy(5))&           segment from n(1) to n(2), opening a
    -alen(xx(3),xx(4),yy(3),yy(4))&           space between n(3) and n(4), connect-
    +alen(xx(1),xx(3),yy(1),yy(3))&           ing the segment in the space, and con-
    +alen(xx(2),xx(4),yy(2),yy(4))&           necting n(5) to n(6).
    +alen(xx(5),xx(6),yy(5),yy(6))
END SUBROUTINE trncst
```

```
SUBROUTINE trnspt(iorder,n)
IMPLICIT NONE
```

```
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
```

This routine does the actual path transport, once `metrop` has approved. `iorder` is an input array giving the present itinerary. The array `n` has as its six elements the beginning `n(1)` and end `n(2)` of the path to be transported, the adjacent cities `n(3)` and `n(4)` between which the path is to be placed, and the cities `n(5)` and `n(6)` that precede and follow the path. `n(4)`, `n(5)`, and `n(6)` are calculated by subroutine `trncst`. On output, `iorder` is modified to reflect the movement of the path segment.

```
INTEGER(I4B) :: m1,m2,m3,nn,ncity
INTEGER(I4B), DIMENSION(size(iorder)) :: jorder
ncity=size(iorder)
m1=1+mod((n(2)-n(1)+ncity),ncity)          Find number of cities from n(1) to n(2) ...
m2=1+mod((n(5)-n(4)+ncity),ncity)          ... and the number from n(4) to n(5)
m3=1+mod((n(3)-n(6)+ncity),ncity)          ... and the number from n(6) to n(3).
jorder(1:m1)=iorder(1+mod((arth(1,1,m1)+n(1)-2),ncity))     Copy the chosen segment.
nn=m1
jorder(nn+1:nn+m2)=iorder(1+mod((arth(1,1,m2)+n(4)-2),ncity))
```

Then copy the segment from `n(4)` to `n(5)`.

```
nn=nn+m2
jorder(nn+1:nn+m3)=iorder(1+mod((arth(1,1,m3)+n(6)-2),ncity))
```

Finally, the segment from `n(6)` to `n(3)`.

```
iorder(1:ncity)=jorder(1:ncity)            Copy jorder back into iorder.
END SUBROUTINE trnspt
END SUBROUTINE anneal
```

⋆ ⋆ ⋆

```
SUBROUTINE amebsa(p,y,pb,yb,ftol,func,iter,temptr)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,iminloc,swap
USE nr, ONLY : ran1
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: iter
REAL(SP), INTENT(INOUT) :: yb
REAL(SP), INTENT(IN) :: ftol,temptr
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y,pb
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: NMAX=200
```

Minimization of the N-dimensional function `func` by simulated annealing combined with the downhill simplex method of Nelder and Mead. The $(N+1) \times N$ matrix `p` is input. Its $N+1$ rows are N-dimensional vectors that are the vertices of the starting simplex. Also input is the vector `y` of length $N+1$, whose components must be preinitialized to the values of `func` evaluated at the $N+1$ vertices (rows) of `p`; `ftol`, the fractional convergence tolerance to be achieved in the function value for an early return; `iter`, and `temptr`. The routine makes `iter` function evaluations at an annealing temperature `temptr`, then returns. You should then decrease `temptr` according to your annealing schedule, reset `iter`, and call the routine again (leaving other arguments unaltered between calls). If `iter` is returned with a positive value, then early convergence and return occurred. If you initialize `yb` to a very large value on the first call, then `yb` and `pb` (an array of length N) will subsequently return the best function value and point ever encountered (even if it is no longer a point in the simplex).

```
INTEGER(I4B) :: ihi,ndim                   Global variables.
REAL(SP) :: yhi
REAL(SP), DIMENSION(size(p,2)) :: psum
call amebsa_private
```

```
CONTAINS

SUBROUTINE amebsa_private
INTEGER(I4B) :: i,ilo,inhi
REAL(SP) :: rtol,ylo,ynhi,ysave,ytry
REAL(SP), DIMENSION(size(y)) :: yt,harvest
ndim=assert_eq(size(p,2),size(p,1)-1,size(y)-1,size(pb),'amebsa')
psum(:)=sum(p(:,:),dim=1)
do                                          Iteration loop.
    call ran1(harvest)
    yt(:)=y(:)-temptr*log(harvest)
      Whenever we "look at" a vertex, it gets a random thermal fluctuation.
    ilo=iminloc(yt(:))                      Determine which point is the highest (worst),
    ylo=yt(ilo)                                 next-highest, and lowest (best).
    ihi=imaxloc(yt(:))
    yhi=yt(ihi)
    yt(ihi)=ylo
    inhi=imaxloc(yt(:))
    ynhi=yt(inhi)
    rtol=2.0_sp*abs(yhi-ylo)/(abs(yhi)+abs(ylo))
      Compute the fractional range from highest to lowest and return if satisfactory.
    if (rtol < ftol .or. iter < 0) then     If returning, put best point and value in
        call swap(y(1),y(ilo))                  slot 1.
        call swap(p(1,:),p(ilo,:))
        RETURN
    end if
      Begin a new iteration. First extrapolate by a factor −1 through the face of the simplex
      across from the high point, i.e., reflect the simplex from the high point.
    ytry=amotsa(-1.0_sp)
    iter=iter-1
    if (ytry <= ylo) then                   Gives a result better than the best point, so
        ytry=amotsa(2.0_sp)                     try an additional extrapolation by a fac-
        iter=iter-1                             tor of 2.
    else if (ytry >= ynhi) then             The reflected point is worse than the second-
        ysave=yhi                               highest, so look for an intermediate lower
        ytry=amotsa(0.5_sp)                     point, i.e., do a one-dimensional contrac-
        iter=iter-1                             tion.
        if (ytry >= ysave) then
              Can't seem to get rid of that high point. Better contract around the lowest
              (best) point.
            p(:,:)=0.5_sp*(p(:,:)+spread(p(ilo,:),1,size(p,1)))
            do i=1,ndim+1
                if (i /= ilo) y(i)=func(p(i,:))
            end do
            iter=iter-ndim                  Keep track of function evaluations.
            psum(:)=sum(p(:,:),dim=1)
        end if
    end if
end do
END SUBROUTINE amebsa_private

FUNCTION amotsa(fac)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fac
REAL(SP) :: amotsa
   Extrapolates by a factor fac through the face of the simplex across from the high point,
   tries it, and replaces the high point if the new point is better.
REAL(SP) :: fac1,fac2,yflu,ytry,harv
REAL(SP), DIMENSION(size(p,2)) :: ptry
fac1=(1.0_sp-fac)/ndim
fac2=fac1-fac
ptry(:)=psum(:)*fac1-p(ihi,:)*fac2
ytry=func(ptry)
if (ytry <= yb) then                        Save the best-ever.
    pb(:)=ptry(:)
```

```
    yb=ytry
end if
call ran1(harv)
yflu=ytry+temptr*log(harv)
if (yflu < yhi) then
    y(ihi)=ytry
    yhi=yflu
    psum(:)=psum(:)-p(ihi,:)+ptry(:)
    p(ihi,:)=ptry(:)
end if
amotsa=yflu
END FUNCTION amotsa
END SUBROUTINE amebsa
```

We *added* a thermal fluctuation to all the current vertices, but we *subtract* it here, so as to give the simplex a thermal Brownian motion: It *likes* to accept any suggested change.

See the discussion of amoeba on p. 1209 for why the routine is coded this way.

Chapter B11. Eigensystems

```
SUBROUTINE jacobi(a,d,v,nrot)
USE nrtype; USE nrutil, ONLY : assert_eq,get_diag,nrerror,unit_matrix,&
    upper_triangle
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: nrot
REAL(SP), DIMENSION(:), INTENT(OUT) :: d
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
```

Computes all eigenvalues and eigenvectors of a real symmetric $N \times N$ matrix a. On output, elements of a above the diagonal are destroyed. d is a vector of length N that returns the eigenvalues of a. v is an $N \times N$ matrix whose columns contain, on output, the normalized eigenvectors of a. nrot returns the number of Jacobi rotations that were required.

```
INTEGER(I4B) :: i,ip,iq,n
REAL(SP) :: c,g,h,s,sm,t,tau,theta,tresh
REAL(SP), DIMENSION(size(d)) :: b,z
n=assert_eq((/size(a,1),size(a,2),size(d),size(v,1),size(v,2)/),'jacobi')
call unit_matrix(v(:,:))                          Initialize v to the identity matrix.
b(:)=get_diag(a(:,:))                             Initialize b and d to the diagonal of
d(:)=b(:)                                             a.
z(:)=0.0                                          This vector will accumulate terms of
nrot=0                                                the form $ta_{pq}$ as in eq. (11.1.14).
do i=1,50
    sm=sum(abs(a),mask=upper_triangle(n,n))       Sum off-diagonal elements.
    if (sm == 0.0) RETURN
      The normal return, which relies on quadratic convergence to machine underflow.
    tresh=merge(0.2_sp*sm/n**2,0.0_sp, i < 4 )
      On the first three sweeps, we will rotate only if tresh exceeded.
    do ip=1,n-1
        do iq=ip+1,n
            g=100.0_sp*abs(a(ip,iq))
              After four sweeps, skip the rotation if the off-diagonal element is small.
            if ((i > 4) .and. (abs(d(ip))+g == abs(d(ip))) &
                .and. (abs(d(iq))+g == abs(d(iq)))) then
                a(ip,iq)=0.0
            else if (abs(a(ip,iq)) > tresh) then
                h=d(iq)-d(ip)
                if (abs(h)+g == abs(h)) then
                    t=a(ip,iq)/h                  t = 1/(2θ)
                else
                    theta=0.5_sp*h/a(ip,iq)       Equation (11.1.10).
                    t=1.0_sp/(abs(theta)+sqrt(1.0_sp+theta**2))
                    if (theta < 0.0) t=-t
                end if
                c=1.0_sp/sqrt(1+t**2)
                s=t*c
                tau=s/(1.0_sp+c)
                h=t*a(ip,iq)
                z(ip)=z(ip)-h
                z(iq)=z(iq)+h
                d(ip)=d(ip)-h
                d(iq)=d(iq)+h
                a(ip,iq)=0.0
```

```
                call jrotate(a(1:ip-1,ip),a(1:ip-1,iq))
                  Case of rotations 1 ≤ j < p.
                call jrotate(a(ip,ip+1:iq-1),a(ip+1:iq-1,iq))
                  Case of rotations p < j < q.
                call jrotate(a(ip,iq+1:n),a(iq,iq+1:n))
                  Case of rotations q < j ≤ n.
                call jrotate(v(:,ip),v(:,iq))
                nrot=nrot+1
            end if
        end do
    end do
    b(:)=b(:)+z(:)
    d(:)=b(:)                                   Update d with the sum of t a_pq,
    z(:)=0.0                                    and reinitialize z.
end do
call nrerror('too many iterations in jacobi')
CONTAINS
SUBROUTINE jrotate(a1,a2)
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a1,a2
REAL(SP), DIMENSION(size(a1)) :: wk1
wk1(:)=a1(:)
a1(:)=a1(:)-s*(a2(:)+a1(:)*tau)
a2(:)=a2(:)+s*(wk1(:)-a2(:)*tau)
END SUBROUTINE jrotate
END SUBROUTINE jacobi
```

As discussed in Volume 1, `jacobi` is generally not competitive with `tqli` in terms of efficiency. However, `jacobi` can be parallelized whereas `tqli` uses an intrinsically serial algorithm. The version of `jacobi` implemented here is likely to be adequate for a small-scale parallel (SSP) machine, but is probably still not competitive with `tqli`. For a massively multiprocessor (MMP) machine, the order of the rotations needs to be chosen in a more complicated pattern than here so that the rotations can be executed in parallel. In this case the Jacobi algorithm may well turn out to be the method of choice. Parallel replacements for `tqli` based on a divide and conquer algorithm have also been proposed. See the discussion after `tqli` on p. 1229.

`call unit_matrix...b(:)=get_diag...` These routines in `nrutil` both require access to the diagonal of a matrix, an operation that is not conveniently provided for in Fortran 90. We have split them off into `nrutil` in case your compiler provides parallel library routines so you can replace our standard versions.

`sm=sum(abs(a),mask=upper_triangle(n,n))` The `upper_triangle` function in `nrutil` returns an upper triangular logical mask. As used here, the mask is true everywhere in the upper triangle of an `n` × `n` matrix, excluding the diagonal. An optional integer argument `extra` allows additional diagonals to be set to true. With `extra=1` the upper triangle including the diagonal would be true. By using the mask, we can conveniently sum over the desired matrix elements in parallel.

`SUBROUTINE jrotate(a1,a2)` This internal subroutine also uses the values of `s` and `tau` from the calling subroutine `jacobi`. Variables in the calling routine are visible to an internal subprogram, but you should be circumspect in making use of this fact. It is easy to overwrite a value in the calling program inadvertently, and it is

often difficult to figure out the logic of an internal routine if not all its variables are declared explicitly. However, jrotate is so simple that there is no danger here.

⋆ ⋆ ⋆

```
SUBROUTINE eigsrt(d,v)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: v
    Given the eigenvalues d and eigenvectors v as output from jacobi (§11.1) or tqli (§11.3),
    this routine sorts the eigenvalues into descending order, and rearranges the columns of v
    correspondingly. The method is straight insertion.
INTEGER(I4B) :: i,j,n
n=assert_eq(size(d),size(v,1),size(v,2),'eigsrt')
do i=1,n-1
    j=imaxloc(d(i:n))+i-1
    if (j /= i) then
        call swap(d(i),d(j))
        call swap(v(:,i),v(:,j))
    end if
end do
END SUBROUTINE eigsrt
```

f90 j=imaxloc... See discussion of imaxloc on p. 1017.

call swap... See discussion of overloaded versions of swap after amoeba on p. 1210.

⋆ ⋆ ⋆

```
SUBROUTINE tred2(a,d,e,novectors)
USE nrtype; USE nrutil, ONLY : assert_eq,outerprod
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: d,e
LOGICAL(LGT), OPTIONAL, INTENT(IN) :: novectors
    Householder reduction of a real, symmetric, N × N matrix a. On output, a is replaced
    by the orthogonal matrix Q effecting the transformation. d returns the diagonal elements
    of the tridiagonal matrix, and e the off-diagonal elements, with e(1)=0. If the optional
    argument novectors is present, only eigenvalues are to be found subsequently, in which
    case a contains no useful information on output.
INTEGER(I4B) :: i,j,l,n
REAL(SP) :: f,g,h,hh,scale
REAL(SP), DIMENSION(size(a,1)) :: gg
LOGICAL(LGT), SAVE :: yesvec=.true.
n=assert_eq(size(a,1),size(a,2),size(d),size(e),'tred2')
if (present(novectors)) yesvec=.not. novectors
do i=n,2,-1
    l=i-1
    h=0.0
    if (l > 1) then
        scale=sum(abs(a(i,1:l)))
        if (scale == 0.0) then                    Skip transformation.
            e(i)=a(i,l)
        else
            a(i,1:l)=a(i,1:l)/scale               Use scaled a's for transformation.
            h=sum(a(i,1:l)**2)                    Form σ in h.
```

```
            f=a(i,l)
            g=-sign(sqrt(h),f)
            e(i)=scale*g
            h=h-f*g                                  Now h is equation (11.2.4).
            a(i,l)=f-g                               Store u in the ith row of a.
            if (yesvec) a(1:l,i)=a(i,1:l)/h          Store u/H in ith column of a.
            do j=1,l                                 Store elements of p in temporarily
                e(j)=(dot_product(a(j,1:j),a(i,1:j)) &     unused elements of e.
                +dot_product(a(j+1:l,j),a(i,j+1:l)))/h
            end do
            f=dot_product(e(1:l),a(i,1:l))
            hh=f/(h+h)                               Form K, equation (11.2.11).
            e(1:l)=e(1:l)-hh*a(i,1:l)
              Form q and store in e overwriting p.
            do j=1,l                                 Reduce a, equation (11.2.13).
                a(j,1:j)=a(j,1:j)-a(i,j)*e(1:j)-e(j)*a(i,1:j)
            end do
        end if
    else
        e(i)=a(i,l)
    end if
    d(i)=h
end do
if (yesvec) d(1)=0.0
e(1)=0.0
do i=1,n                                             Begin accumulation of transforma-
    if (yesvec) then                                     tion matrices.
        l=i-1
        if (d(i) /= 0.0) then
              This block skipped when i=1. Use u and u/H stored in a to form P · Q.
            gg(1:l)=matmul(a(i,1:l),a(1:l,1:l))
            a(1:l,1:l)=a(1:l,1:l)-outerprod(a(1:l,i),gg(1:l))
        end if
        d(i)=a(i,i)
        a(i,i)=1.0                                   Reset row and column of a to iden-
        a(i,1:l)=0.0                                     tity matrix for next iteration.
        a(1:l,i)=0.0
    else
        d(i)=a(i,i)
    end if
end do
END SUBROUTINE tred2
```

f90 This routine gives a nice example of the usefulness of optional arguments. The routine is written under the assumption that usually you will want to find both eigenvalues and eigenvectors. In this case you just supply the arguments a, d, and e. If, however, you want only eigenvalues, you supply the additional logical argument `novectors` with the value `.true.`. The routine then skips the unnecessary computations. Supplying `novectors` with the value `.false.` has the same effect as omitting it.

⋆ ⋆ ⋆

```
SUBROUTINE tqli(d,e,z)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : pythag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d,e
REAL(SP), DIMENSION(:,:), OPTIONAL, INTENT(INOUT) :: z
```

QL algorithm with implicit shifts, to determine the eigenvalues and eigenvectors of a real, symmetric, tridiagonal matrix, or of a real, symmetric matrix previously reduced by `tred2`

§11.2. d is a vector of length N. On input, its elements are the diagonal elements of the tridiagonal matrix. On output, it returns the eigenvalues. The vector e inputs the subdiagonal elements of the tridiagonal matrix, with e(1) arbitrary. On output e is destroyed. When finding only the eigenvalues, the optional argument z is omitted. If the eigenvectors of a tridiagonal matrix are desired, the $N \times N$ matrix z is input as the identity matrix. If the eigenvectors of a matrix that has been reduced by tred2 are required, then z is input as the matrix output by tred2. In either case, the kth column of z returns the normalized eigenvector corresponding to d(k).

```
INTEGER(I4B) :: i,iter,l,m,n,ndum
REAL(SP) :: b,c,dd,f,g,p,r,s
REAL(SP), DIMENSION(size(e)) :: ff
n=assert_eq(size(d),size(e),'tqli: n')
if (present(z)) ndum=assert_eq(n,size(z,1),size(z,2),'tqli: ndum')
e(:)=eoshift(e(:),1)                          Convenient to renumber the elements of
do l=1,n                                          e.
    iter=0
    iterate: do
        do m=l,n-1                            Look for a single small subdiagonal ele-
            dd=abs(d(m))+abs(d(m+1))              ment to split the matrix.
            if (abs(e(m))+dd == dd) exit
        end do
        if (m == l) exit iterate
        if (iter == 30) call nrerror('too many iterations in tqli')
        iter=iter+1
        g=(d(l+1)-d(l))/(2.0_sp*e(l))         Form shift.
        r=pythag(g,1.0_sp)
        g=d(m)-d(l)+e(l)/(g+sign(r,g))        This is d_m − k_s.
        s=1.0
        c=1.0
        p=0.0
        do i=m-1,l,-1                         A plane rotation as in the original QL,
            f=s*e(i)                              followed by Givens rotations to re-
            b=c*e(i)                              store tridiagonal form.
            r=pythag(f,g)
            e(i+1)=r
            if (r == 0.0) then                Recover from underflow.
                d(i+1)=d(i+1)-p
                e(m)=0.0
                cycle iterate
            end if
            s=f/r
            c=g/r
            g=d(i+1)-p
            r=(d(i)-g)*s+2.0_sp*c*b
            p=s*r
            d(i+1)=g+p
            g=c*r-b
            if (present(z)) then              Form eigenvectors.
                ff(1:n)=z(1:n,i+1)
                z(1:n,i+1)=s*z(1:n,i)+c*ff(1:n)
                z(1:n,i)=c*z(1:n,i)-s*ff(1:n)
            end if
        end do
        d(l)=d(l)-p
        e(l)=g
        e(m)=0.0
    end do iterate
end do
END SUBROUTINE tqli
```

The routine tqli is intrinsically serial. A parallel replacement based on a divide and conquer algorithm has been proposed [1,2]. The idea is to split the tridiagonal matrix recursively into two tridiagonal matrices of

half the size plus a correction. Given the eigensystems of the two smaller tridiagonal matrices, it is possible to join them together and add in the effect of the correction. When some small size of tridiagonal matrix is reached during the recursive splitting, its eigensystem is found directly with a routine like `tqli`. Each of these small problems is independent and can be assigned to an independent processor. The procedures for sewing together can also be done independently. For very large matrices, this algorithm can be an order of magnitude faster than `tqli` even on a serial machine, and no worse than a factor of 2 or 3 slower, depending on the matrix. Unfortunately the parallelism is not well expressed in Fortran 90. Also, the sewing together requires quite involved coding. For an implementation see the LAPACK routine SSTEDC. Another parallel strategy for eigensystems uses inverse iteration, where each eigenvalue and eigenvector can be found independently [3].

This routine uses `z` as an optional argument that is required only if eigenvectors are being found as well as eigenvalues.

`iterate: do` See discussion of named do loops after `simplx` on p. 1219.

⋆ ⋆ ⋆

```
SUBROUTINE balanc(a)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
REAL(SP), PARAMETER :: RADX=radix(a),SQRADX=RADX**2
```

Given an $N \times N$ matrix `a`, this routine replaces it by a balanced matrix with identical eigenvalues. A symmetric matrix is already balanced and is unaffected by this procedure. The parameter `RADX` is the machine's floating-point radix.

```
INTEGER(I4B) :: i,last,ndum
REAL(SP) :: c,f,g,r,s
ndum=assert_eq(size(a,1),size(a,2),'balanc')
do
    last=1
    do i=1,size(a,1)                          Calculate row and column norms.
        c=sum(abs(a(:,i)))-a(i,i)
        r=sum(abs(a(i,:)))-a(i,i)
        if (c /= 0.0 .and. r /= 0.0) then     If both are nonzero,
            g=r/RADX
            f=1.0
            s=c+r
            do                                find the integer power of the ma-
                if (c >= g) exit                  chine radix that comes closest to
                f=f*RADX                          balancing the matrix.
                c=c*SQRADX
            end do
            g=r*RADX
            do
                if (c <= g) exit
                f=f/RADX
                c=c/SQRADX
            end do
            if ((c+r)/f < 0.95_sp*s) then
                last=0
                g=1.0_sp/f
                a(i,:)=a(i,:)*g               Apply similarity transformation.
                a(:,i)=a(:,i)*f
            end if
        end if
```

```
    end do
    if (last /= 0) exit
end do
END SUBROUTINE balanc
```

f90 `REAL(SP), PARAMETER :: RADX=radix(a)...` Fortran 90 provides a nice collection of numeric inquiry intrinsic functions. Here we find the machine's floating-point radix. Note that only the type of the argument `a` affects the returned function value.

⋆ ⋆ ⋆

```
SUBROUTINE elmhes(a)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,outerprod,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
```

Reduction to Hessenberg form by the elimination method. The real, nonsymmetric, $N \times N$ matrix `a` is replaced by an upper Hessenberg matrix with identical eigenvalues. Recommended, but not required, is that this routine be preceded by `balanc`. On output, the Hessenberg matrix is in elements `a`(i, j) with $i \leq j + 1$. Elements with $i > j + 1$ are to be thought of as zero, but are returned with random values.

```
INTEGER(I4B) :: i,m,n
REAL(SP) :: x
REAL(SP), DIMENSION(size(a,1)) :: y
n=assert_eq(size(a,1),size(a,2),'elmhes')
do m=2,n-1                                  m is called r + 1 in the text.
    i=imaxloc(abs(a(m:n,m-1)))+m-1          Find the pivot.
    x=a(i,m-1)
    if (i /= m) then                        Interchange rows and columns.
        call swap(a(i,m-1:n),a(m,m-1:n))
        call swap(a(:,i),a(:,m))
    end if
    if (x /= 0.0) then                      Carry out the elimination.
        y(m+1:n)=a(m+1:n,m-1)/x
        a(m+1:n,m-1)=y(m+1:n)
        a(m+1:n,m:n)=a(m+1:n,m:n)-outerprod(y(m+1:n),a(m,m:n))
        a(:,m)=a(:,m)+matmul(a(:,m+1:n),y(m+1:n))
    end if
end do
END SUBROUTINE elmhes
```

f90 `y(m+1:n)=...` If the four lines of code starting here were all coded for a serial machine in a single do-loop starting with `do i=m+1,n` (see Volume 1), it would pay to test whether `y` was zero because the next three lines could then be skipped for that value of `i`. There is no convenient way to do this here, even with a `where`, since the shape of the arrays on each of the three lines is different. For a parallel machine it is probably best just to do a few unnecessary multiplies and skip the test for zero values of `y`.

⋆ ⋆ ⋆

```
SUBROUTINE hqr(a,wr,wi)
USE nrtype; USE nrutil, ONLY : assert_eq,diagadd,nrerror,upper_triangle
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: wr,wi
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
```

Finds all eigenvalues of an $N \times N$ upper Hessenberg matrix a. On input a can be exactly as output from elmhes §11.5; on output it is destroyed. The real and imaginary parts of the N eigenvalues are returned in wr and wi, respectively.

```
INTEGER(I4B) :: i,its,k,l,m,n,nn,mnnk
REAL(SP) :: anorm,p,q,r,s,t,u,v,w,x,y,z
REAL(SP), DIMENSION(size(a,1)) :: pp
n=assert_eq(size(a,1),size(a,2),size(wr),size(wi),'hqr')
anorm=sum(abs(a),mask=upper_triangle(n,n,extra=2))
```

Compute matrix norm for possible use in locating single small subdiagonal element.

```
nn=n
t=0.0                                   Gets changed only by an exceptional shift.
do                                      Begin search for next eigenvalue: "Do while
    if (nn < 1) exit                        nn >= 1".
    its=0
    iterate: do                         Begin iteration.
        do l=nn,2,-1                    Look for single small subdiagonal element.
            s=abs(a(l-1,l-1))+abs(a(l,l))
            if (s == 0.0) s=anorm
            if (abs(a(l,l-1))+s == s) exit
        end do
        x=a(nn,nn)
        if (l == nn) then               One root found.
            wr(nn)=x+t
            wi(nn)=0.0
            nn=nn-1
            exit iterate                Go back for next eigenvalue.
        end if
        y=a(nn-1,nn-1)
        w=a(nn,nn-1)*a(nn-1,nn)
        if (l == nn-1) then             Two roots found ...
            p=0.5_sp*(y-x)
            q=p**2+w
            z=sqrt(abs(q))
            x=x+t
            if (q >= 0.0) then          ... a real pair ...
                z=p+sign(z,p)
                wr(nn)=x+z
                wr(nn-1)=wr(nn)
                if (z /= 0.0) wr(nn)=x-w/z
                wi(nn)=0.0
                wi(nn-1)=0.0
            else                        ... a complex pair.
                wr(nn)=x+p
                wr(nn-1)=wr(nn)
                wi(nn)=z
                wi(nn-1)=-z
            end if
            nn=nn-2
            exit iterate                Go back for next eigenvalue.
        end if
          No roots found. Continue iteration.
        if (its == 30) call nrerror('too many iterations in hqr')
        if (its == 10 .or. its == 20) then      Form exceptional shift.
            t=t+x
            call diagadd(a(1:nn,1:nn),-x)
            s=abs(a(nn,nn-1))+abs(a(nn-1,nn-2))
            x=0.75_sp*s
            y=x
            w=-0.4375_sp*s**2
```

```
end if
its=its+1
do m=nn-2,l,-1                          Form shift and then look for 2 consecu-
    z=a(m,m)                                tive small subdiagonal elements.
    r=x-z
    s=y-z
    p=(r*s-w)/a(m+1,m)+a(m,m+1)         Equation (11.6.23).
    q=a(m+1,m+1)-z-r-s
    r=a(m+2,m+1)
    s=abs(p)+abs(q)+abs(r)              Scale to prevent overflow or underflow.
    p=p/s
    q=q/s
    r=r/s
    if (m == l) exit
    u=abs(a(m,m-1))*(abs(q)+abs(r))
    v=abs(p)*(abs(a(m-1,m-1))+abs(z)+abs(a(m+1,m+1)))
    if (u+v == v) exit                  Equation (11.6.26).
end do
do i=m+2,nn
    a(i,i-2)=0.0
    if (i /= m+2) a(i,i-3)=0.0
end do
do k=m,nn-1                             Double QR step on rows l to nn and
    if (k /= m) then                        columns m to nn.
        p=a(k,k-1)                      Begin setup of Householder vector.
        q=a(k+1,k-1)
        r=0.0
        if (k /= nn-1) r=a(k+2,k-1)
        x=abs(p)+abs(q)+abs(r)
        if (x /= 0.0) then
            p=p/x                       Scale to prevent overflow or underflow.
            q=q/x
            r=r/x
        end if
    end if
    s=sign(sqrt(p**2+q**2+r**2),p)
    if (s /= 0.0) then
        if (k == m) then
            if (l /= m) a(k,k-1)=-a(k,k-1)
        else
            a(k,k-1)=-s*x
        end if
        p=p+s                           Equations (11.6.24).
        x=p/s
        y=q/s
        z=r/s
        q=q/p
        r=r/p                           Ready for row modification.
        pp(k:nn)=a(k,k:nn)+q*a(k+1,k:nn)
        if (k /= nn-1) then
            pp(k:nn)=pp(k:nn)+r*a(k+2,k:nn)
            a(k+2,k:nn)=a(k+2,k:nn)-pp(k:nn)*z
        end if
        a(k+1,k:nn)=a(k+1,k:nn)-pp(k:nn)*y
        a(k,k:nn)=a(k,k:nn)-pp(k:nn)*x
        mnnk=min(nn,k+3)                Column modification.
        pp(l:mnnk)=x*a(l:mnnk,k)+y*a(l:mnnk,k+1)
        if (k /= nn-1) then
            pp(l:mnnk)=pp(l:mnnk)+z*a(l:mnnk,k+2)
            a(l:mnnk,k+2)=a(l:mnnk,k+2)-pp(l:mnnk)*r
        end if
        a(l:mnnk,k+1)=a(l:mnnk,k+1)-pp(l:mnnk)*q
        a(l:mnnk,k)=a(l:mnnk,k)-pp(l:mnnk)
    end if
```

```
        end do
    end do iterate                                  Go back for next iteration on current eigen-
end do                                                  value.
END SUBROUTINE hqr
```

`anorm=sum(abs(a),mask=upper_triangle(n,n,extra=2))` See the discussion of `upper_triangle` after `jacobi` on p. 1226. Setting `extra=2` here picks out the upper Hessenberg part of the matrix.

`iterate: do` We use a named loop to improve the readability and structuring of the routine. The if-blocks that test for one or two roots end with `exit iterate`, transferring control back to the outermost loop and thus starting a search for the next root.

`call diagadd...` The routines that operate on the diagonal of a matrix are collected in `nrutil` partly so you can write clear code and partly in the hope that compiler writers will provide parallel library routines. Fortran 90 does not provide convenient parallel access to the diagonal of a matrix.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.6 and references therein. [1]

Sorensen, D.C., and Tang, P.T.P. 1991, *SIAM Journal on Numerical Analysis*, vol. 28, pp. 1752–1775. [2]

Lo, S.-S., Philippe, B., and Sameh, A. 1987, *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. s155–s165. [3]

Chapter B12. Fast Fourier Transform

The algorithms underlying the parallel routines in this chapter are described in §22.4. As described there, the basic building block is a routine for simultaneously taking the FFT of each row of a two-dimensional matrix:

```
SUBROUTINE fourrow_sp(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces each row (constant first index) of `data(1:`M`,1:`N`)` by its discrete Fourier transform (transform on second index), if `isign` is input as 1; or replaces each row of `data` by N times its inverse discrete Fourier transform, if `isign` is input as -1. N must be an integer power of 2. Parallelism is M-fold on the first index of `data`.

```
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,1)) :: temp
COMPLEX(DPC) :: w,wp                    Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,2)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_sp')
n2=n/2
j=n2
```

This is the bit-reversal section of the routine.

```
do i=1,n-2
    if (j > i) call swap(data(:,j+1),data(:,i+1))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
```

Here begins the Danielson-Lanczos section of the routine.

```
do                                      Outer loop executed log2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)             Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                         Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(:,j)           This is the Danielson-Lanczos formula.
            data(:,j)=data(:,i)-temp
            data(:,i)=data(:,i)+temp
        end do
        w=w*wp+w                        Trigonometric recurrence.
    end do
    mmax=istep
```

```
end do
END SUBROUTINE fourrow_sp
```

f90 `call assert(iand(n,n-1)==0 ...` All the Fourier routines in this chapter require the dimension N of the data to be a power of 2. This is easily tested for by AND'ing N and $N-1$: N should have the binary representation $10000\ldots$, in which case $N-1 = 01111\ldots$.

```
SUBROUTINE fourrow_dp(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(DPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(DPC), DIMENSION(size(data,1)) :: temp
COMPLEX(DPC) :: w,wp
COMPLEX(DPC) :: ws
n=size(data,2)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_dp')
n2=n/2
j=n2
do i=1,n-2
    if (j > i) call swap(data(:,j+1),data(:,i+1))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
do
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(:,j)
            data(:,j)=data(:,i)-temp
            data(:,i)=data(:,i)+temp
        end do
        w=w*wp+w
    end do
    mmax=istep
end do
END SUBROUTINE fourrow_dp
```

```
SUBROUTINE fourrow_3d(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

If `isign` is input as 1, replaces each third-index section (constant first and second indices) of `data(1:`L`,1:`M`,1:`N`)` by its discrete Fourier transform (transform on third index); or

replaces each third-index section of `data` by N times its inverse discrete Fourier transform, if `isign` is input as -1. N must be an integer power of 2. Parallelism is $L \times M$-fold on the first and second indices of `data`.

```
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,1),size(data,2)) :: temp
COMPLEX(DPC) :: w,wp                    Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,3)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourrow_3d')
n2=n/2
j=n2
```

This is the bit-reversal section of the routine.

```
do i=1,n-2
    if (j > i) call swap(data(:,:,j+1),data(:,:,i+1))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
```

Here begins the Danielson-Lanczos section of the routine.

```
do                                      Outer loop executed log2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)             Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                         Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(:,:,j)         This is the Danielson-Lanczos formula.
            data(:,:,j)=data(:,:,i)-temp
            data(:,:,i)=data(:,:,i)+temp
        end do
        w=w*wp+w                        Trigonometric recurrence.
    end do
    mmax=istep
end do
END SUBROUTINE fourrow_3d
```

⋆ ⋆ ⋆

Exactly as in the preceding routines, we can take the FFT of each *column* of a two-dimensional matrix, and for each ***first-index*** section of a three-dimensional array.

```
SUBROUTINE fourcol(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces each column (constant second index) of `data(1:`N`,1:`M`)` by its discrete Fourier transform (transform on first index), if `isign` is input as 1; or replaces each row of `data`

by N times its inverse discrete Fourier transform, if `isign` is input as -1. N must be an integer power of 2. Parallelism is M-fold on the second index of `data`.

```
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,2)) :: temp
COMPLEX(DPC) :: w,wp                    Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,1)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourcol')
n2=n/2
j=n2
```

This is the bit-reversal section of the routine.

```
do i=1,n-2
    if (j > i) call swap(data(j+1,:),data(i+1,:))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
```

Here begins the Danielson-Lanczos section of the routine.

```
do                                      Outer loop executed log_2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)             Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                         Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(j,:)           This is the Danielson-Lanczos formula.
            data(j,:)=data(i,:)-temp
            data(i,:)=data(i,:)+temp
        end do
        w=w*wp+w                        Trigonometric recurrence.
    end do
    mmax=istep
end do
END SUBROUTINE fourcol
```

```
SUBROUTINE fourcol_3d(data,isign)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

If `isign` is input as 1, replaces each first-index section (constant second and third indices) of `data`$(1{:}N,1{:}M,1{:}L)$ by its discrete Fourier transform (transform on first index); or replaces each first-index section of `data` by N times its inverse discrete Fourier transform, if `isign` is input as -1. N must be an integer power of 2. Parallelism is $M \times L$-fold on the second and third indices of `data`.

```
INTEGER(I4B) :: n,i,istep,j,m,mmax,n2
REAL(DP) :: theta
COMPLEX(SPC), DIMENSION(size(data,2),size(data,3)) :: temp
COMPLEX(DPC) :: w,wp                    Double precision for the trigonometric recurrences.
COMPLEX(SPC) :: ws
n=size(data,1)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in fourcol_3d')
n2=n/2
```

```
j=n2
  This is the bit-reversal section of the routine.
do i=1,n-2
    if (j > i) call swap(data(j+1,:,:),data(i+1,:,:))
    m=n2
    do
        if (m < 2 .or. j < m) exit
        j=j-m
        m=m/2
    end do
    j=j+m
end do
mmax=1
  Here begins the Danielson-Lanczos section of the routine.
do                                          Outer loop executed log2 N times.
    if (n <= mmax) exit
    istep=2*mmax
    theta=PI_D/(isign*mmax)                 Initialize for the trigonometric recurrence.
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
    w=cmplx(1.0_dp,0.0_dp,kind=dpc)
    do m=1,mmax                             Here are the two nested inner loops.
        ws=w
        do i=m,n,istep
            j=i+mmax
            temp=ws*data(j,:,:)             This is the Danielson-Lanczos formula.
            data(j,:,:)=data(i,:,:)-temp
            data(i,:,:)=data(i,:,:)+temp
        end do
        w=w*wp+w                            Trigonometric recurrence.
    end do
    mmax=istep
end do
END SUBROUTINE fourcol_3d
```

⋆ ⋆ ⋆

Here now are implementations of the method of §22.4 for the FFT of one-dimensional single- and double-precision complex arrays:

```
SUBROUTINE four1_sp(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a complex array `data` by its discrete Fourier transform, if `isign` is input as 1; or replaces `data` by its inverse discrete Fourier transform times the size of `data`, if `isign` is input as −1. The size of `data` must be an integer power of 2. Parallelism is achieved by internally reshaping the input array to two dimensions. (Use this version if `fourrow` is faster than `fourcol` on your machine.)

```
COMPLEX(SPC), DIMENSION(:,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_sp')
  Find dimensions as close to square as possible, allocate space, and reshape the input array.
m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
call fourrow(dat,isign)                     Transform on second index.
```

```
theta=arth(0,isign,m1)*TWOPI_D/n        Set up recurrence.
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
w=cmplx(1.0_dp,0.0_dp,kind=dpc)
do j=2,m2                               Multiply by the extra phase factor.
    w=w*wp+w
    dat(:,j)=dat(:,j)*w
end do
temp=transpose(dat)                     Transpose, and transform on (original) first in-
call fourrow(temp,isign)                    dex.
data=reshape(temp,shape(data))          Reshape the result back to one dimension.
deallocate(dat,w,wp,theta,temp)
END SUBROUTINE four1_sp
```

```
SUBROUTINE four1_dp(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(DPC), DIMENSION(:,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_dp')
m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
call fourrow(dat,isign)
theta=arth(0,isign,m1)*TWOPI_D/n
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
w=cmplx(1.0_dp,0.0_dp,kind=dpc)
do j=2,m2
    w=w*wp+w
    dat(:,j)=dat(:,j)*w
end do
temp=transpose(dat)
call fourrow(temp,isign)
data=reshape(temp,shape(data))
deallocate(dat,w,wp,theta,temp)
END SUBROUTINE four1_dp
```

The above routines use `fourrow` exclusively, on the assumption that it is faster than its sibling `fourcol`. When that is the case (as we typically find), it is likely that `four1_sp` is also faster than Volume 1's scalar `four1`. The reason, on scalar machines, is that `fourrow`'s parallelism is taking better advantage of cache memory locality.

If `fourrow` is *not* faster than `fourcol` on your machine, then you should instead try the following alternative FFT version that uses `fourcol` only.

```
SUBROUTINE four1_alt(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
USE nr, ONLY : fourcol
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a complex array `data` by its discrete Fourier transform, if `isign` is input as 1; or replaces `data` by its inverse discrete Fourier transform times the size of `data`, if `isign` is

input as -1. The size of data must be an integer power of 2. Parallelism is achieved by internally reshaping the input array to two dimensions. (Use this version *only* if fourcol is faster than fourrow on your machine.)

```
COMPLEX(SPC), DIMENSION(:,:), ALLOCATABLE :: dat,temp
COMPLEX(DPC), DIMENSION(:), ALLOCATABLE :: w,wp
REAL(DP), DIMENSION(:), ALLOCATABLE :: theta
INTEGER(I4B) :: n,m1,m2,j
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_alt')
```

Find dimensions as close to square as possible, allocate space, and reshape the input array.

```
m1=2**ceiling(0.5_sp*log(real(n,sp))/0.693147_sp)
m2=n/m1
allocate(dat(m1,m2),theta(m1),w(m1),wp(m1),temp(m2,m1))
dat=reshape(data,shape(dat))
temp=transpose(dat)                    Transpose and transform on (original) second index.
call fourcol(temp,isign)
theta=arth(0,isign,m1)*TWOPI_D/n       Set up recurrence.
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
w=cmplx(1.0_dp,0.0_dp,kind=dpc)
do j=2,m2                              Multiply by the extra phase factor.
    w=w*wp+w
    temp(j,:)=temp(j,:)*w
end do
dat=transpose(temp)                    Transpose, and transform on (original) first index.
call fourcol(dat,isign)
temp=transpose(dat)                    Transpose and then reshape the result back to one dimension.
data=reshape(temp,shape(data))
deallocate(dat,w,wp,theta,temp)
END SUBROUTINE four1_alt
```

⋆ ⋆ ⋆

With all the machinery of fourrow and fourcol, two-dimensional FFTs are extremely straightforward. Again there is an alternative version provided in case your hardware favors fourcol (which would be, we think, unusual).

```
SUBROUTINE four2(data,isign)
USE nrtype
USE nr, ONLY : fourrow
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a 2-d complex array data by its discrete 2-d Fourier transform, if isign is input as 1; or replaces data by its inverse 2-d discrete Fourier transform times the product of its two sizes, if isign is input as -1. Both of data's sizes must be integer powers of 2 (this is checked for in fourrow). Parallelism is by use of fourrow.

```
COMPLEX(SPC), DIMENSION(size(data,2),size(data,1)) :: temp
call fourrow(data,isign)          Transform in second dimension.
temp=transpose(data)              Tranpose.
call fourrow(temp,isign)          Transform in (original) first dimension.
data=transpose(temp)              Transpose into data.
END SUBROUTINE four2
```

```
SUBROUTINE four2_alt(data,isign)
USE nrtype
USE nr, ONLY : fourcol
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a 2-d complex array `data` by its discrete 2-d Fourier transform, if `isign` is input as 1; or replaces `data` by its inverse 2-d discrete Fourier transform times the product of its two sizes, if `isign` is input as -1. Both of `data`'s sizes must be integer powers of 2 (this is checked for in `fourcol`). Parallelism is by use of `fourcol`. (Use this version *only* if `fourcol` is faster than `fourrow` on your machine.)

```
COMPLEX(SPC), DIMENSION(size(data,2),size(data,1)) :: temp
temp=transpose(data)            Tranpose.
call fourcol(temp,isign)        Transform in (original) second dimension.
data=transpose(temp)            Transpose.
call fourcol(data,isign)        Transform in (original) first dimension.
END SUBROUTINE four2_alt
```

⋆ ⋆ ⋆

Most of the remaining routines in this chapter simply call one or another of the above FFT routines, with a small amount of auxiliary computation, so they are fairly straightforward conversions from their Volume 1 counterparts.

```
SUBROUTINE twofft(data1,data2,fft1,fft2)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: fft1,fft2
```

Given two real input arrays `data1` and `data2` of length N, this routine calls `four1` and returns two complex output arrays, `fft1` and `fft2`, each of complex length N, that contain the discrete Fourier transforms of the respective `data` arrays. N must be an integer power of 2.

```
INTEGER(I4B) :: n,n2
COMPLEX(SPC), PARAMETER :: C1=(0.5_sp,0.0_sp), C2=(0.0_sp,-0.5_sp)
COMPLEX, DIMENSION(size(data1)/2+1) :: h1,h2
n=assert_eq(size(data1),size(data2),size(fft1),size(fft2),'twofft')
call assert(iand(n,n-1)==0, 'n must be a power of 2 in twofft')
fft1=cmplx(data1,data2,kind=spc)      Pack the two real arrays into one complex array.
call four1(fft1,1)                    Transform the complex array.
fft2(1)=cmplx(aimag(fft1(1)),0.0_sp,kind=spc)
fft1(1)=cmplx(real(fft1(1)),0.0_sp,kind=spc)
n2=n/2+1
h1(2:n2)=C1*(fft1(2:n2)+conjg(fft1(n:n2:-1)))      Use symmetries to separate the
h2(2:n2)=C2*(fft1(2:n2)-conjg(fft1(n:n2:-1)))          two transforms.
fft1(2:n2)=h1(2:n2)                   Ship them out in two complex arrays.
fft1(n:n2:-1)=conjg(h1(2:n2))
fft2(2:n2)=h2(2:n2)
fft2(n:n2:-1)=conjg(h2(2:n2))
END SUBROUTINE twofft
```

⋆ ⋆ ⋆

```
SUBROUTINE realft_sp(data,isign,zdata)
USE nrtype; USE nrutil, ONLY : assert,assert_eq,zroots_unity
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(SPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
```

When isign $= 1$, calculates the Fourier transform of a set of N real-valued data points, input in the array data. If the optional argument zdata is not present, the data are replaced by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned as elements data(1) and data(2), respectively. If the complex array zdata of length $N/2$ is present, data is unchanged and the transform is returned in zdata. N must be a power of 2. If isign $= -1$, this routine calculates the inverse transform of a complex data array if it is the transform of real data. (Result in this case must be multiplied by $2/N$.) The data can be supplied either in data, with zdata absent, or in zdata.

```
INTEGER(I4B) :: n,ndum,nh,nq
COMPLEX(SPC), DIMENSION(size(data)/4) :: w
COMPLEX(SPC), DIMENSION(size(data)/4-1) :: h1,h2
COMPLEX(SPC), DIMENSION(:), POINTER :: cdata      Used for internal complex computa-
COMPLEX(SPC) :: z                                    tions.
REAL(SP) :: c1=0.5_sp,c2
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in realft_sp')
nh=n/2
nq=n/4
if (present(zdata)) then
    ndum=assert_eq(n/2,size(zdata),'realft_sp')
    cdata=>zdata                                  Use zdata as cdata.
    if (isign == 1) cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
else
    allocate(cdata(n/2))                          Have to allocate storage ourselves.
    cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
end if
if (isign == 1) then
    c2=-0.5_sp
    call four1(cdata,+1)                          The forward transform is here.
else                                              Otherwise set up for an inverse trans-
    c2=0.5_sp                                        form.
end if
w=zroots_unity(sign(n,isign),n/4)
w=cmplx(-aimag(w),real(w),kind=spc)
h1=c1*(cdata(2:nq)+conjg(cdata(nh:nq+2:-1)))      The two separate transforms are sep-
h2=c2*(cdata(2:nq)-conjg(cdata(nh:nq+2:-1)))         arated out of cdata.
```

Next they are recombined to form the true transform of the original real data:

```
cdata(2:nq)=h1+w(2:nq)*h2
cdata(nh:nq+2:-1)=conjg(h1-w(2:nq)*h2)
z=cdata(1)                                        Squeeze the first and last data to-
if (isign == 1) then                                 gether to get them all within the
    cdata(1)=cmplx(real(z)+aimag(z),real(z)-aimag(z),kind=spc)     original array.
else
    cdata(1)=cmplx(c1*(real(z)+aimag(z)),c1*(real(z)-aimag(z)),kind=spc)
    call four1(cdata,-1)                          This is the inverse transform for the
end if                                               case isign=-1.
if (present(zdata)) then                          Ship out answer in data if required.
    if (isign /= 1) then
        data(1:n-1:2)=real(cdata)
        data(2:n:2)=aimag(cdata)
    end if
else
    data(1:n-1:2)=real(cdata)
    data(2:n:2)=aimag(cdata)
    deallocate(cdata)
end if
```

```
END SUBROUTINE realft_sp



SUBROUTINE realft_dp(data,isign,zdata)
USE nrtype; USE nrutil, ONLY : assert,assert_eq,zroots_unity
USE nr, ONLY : four1
IMPLICIT NONE
REAL(DP), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
COMPLEX(DPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
INTEGER(I4B) :: n,ndum,nh,nq
COMPLEX(DPC), DIMENSION(size(data)/4) :: w
COMPLEX(DPC), DIMENSION(size(data)/4-1) :: h1,h2
COMPLEX(DPC), DIMENSION(:), POINTER :: cdata
COMPLEX(DPC) :: z
REAL(DP) :: c1=0.5_dp,c2
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in realft_dp')
nh=n/2
nq=n/4
if (present(zdata)) then
    ndum=assert_eq(n/2,size(zdata),'realft_dp')
    cdata=>zdata
    if (isign == 1) cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
else
    allocate(cdata(n/2))
    cdata=cmplx(data(1:n-1:2),data(2:n:2),kind=spc)
end if
if (isign == 1) then
    c2=-0.5_dp
    call four1(cdata,+1)
else
    c2=0.5_dp
end if
w=zroots_unity(sign(n,isign),n/4)
w=cmplx(-aimag(w),real(w),kind=dpc)
h1=c1*(cdata(2:nq)+conjg(cdata(nh:nq+2:-1)))
h2=c2*(cdata(2:nq)-conjg(cdata(nh:nq+2:-1)))
cdata(2:nq)=h1+w(2:nq)*h2
cdata(nh:nq+2:-1)=conjg(h1-w(2:nq)*h2)
z=cdata(1)
if (isign == 1) then
    cdata(1)=cmplx(real(z)+aimag(z),real(z)-aimag(z),kind=dpc)
else
    cdata(1)=cmplx(c1*(real(z)+aimag(z)),c1*(real(z)-aimag(z)),kind=dpc)
    call four1(cdata,-1)
end if
if (present(zdata)) then
    if (isign /= 1) then
        data(1:n-1:2)=real(cdata)
        data(2:n:2)=aimag(cdata)
    end if
else
    data(1:n-1:2)=real(cdata)
    data(2:n:2)=aimag(cdata)
    deallocate(cdata)
end if
END SUBROUTINE realft_dp
```

⋆ ⋆ ⋆

```
SUBROUTINE sinft(y)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
```

Calculates the sine transform of a set of N real-valued data points stored in array y. The number N must be a power of 2. On exit y is replaced by its transform. This program, without changes, also calculates the inverse sine transform, but in this case the output array should be multiplied by $2/N$.

```
REAL(SP), DIMENSION(size(y)/2+1) :: wi
REAL(SP), DIMENSION(size(y)/2) :: y1,y2
INTEGER(I4B) :: n,nh
n=size(y)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in sinft')
nh=n/2
wi=aimag(zroots_unity(n+n,nh+1))        Calculate the sine for the auxiliary array.
y(1)=0.0
y1=wi(2:nh+1)*(y(2:nh+1)+y(n:nh+1:-1))
```

Construct the two pieces of the auxiliary array.

```
y2=0.5_sp*(y(2:nh+1)-y(n:nh+1:-1))      Put them together to make the auxiliary array.
y(2:nh+1)=y1+y2
y(n:nh+1:-1)=y1-y2
call realft(y,+1)                       Transform the auxiliary array.
y(1)=0.5_sp*y(1)                        Initialize the sum used for odd terms.
y(2)=0.0
y1=cumsum(y(1:n-1:2))                   Odd terms are determined by this running sum.
y(1:n-1:2)=y(2:n:2)                     Even terms in the transform are determined directly.
y(2:n:2)=y1
END SUBROUTINE sinft
```

```
SUBROUTINE cosft1(y)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
```

Calculates the cosine transform of a set of $N+1$ real-valued data points y. The transformed data replace the original data in array y. N must be a power of 2. This program, without changes, also calculates the inverse cosine transform, but in this case the output array should be multiplied by $2/N$.

```
COMPLEX(SPC), DIMENSION((size(y)-1)/2) :: w
REAL(SP), DIMENSION((size(y)-1)/2-1) :: y1,y2
REAL(SP) :: summ
INTEGER(I4B) :: n,nh
n=size(y)-1
call assert(iand(n,n-1)==0, 'n must be a power of 2 in cosft1')
nh=n/2
w=zroots_unity(n+n,nh)
summ=0.5_sp*(y(1)-y(n+1))
y(1)=0.5_sp*(y(1)+y(n+1))
y1=0.5_sp*(y(2:nh)+y(n:nh+2:-1))        Construct the two pieces of the auxiliary array.
y2=y(2:nh)-y(n:nh+2:-1)
summ=summ+sum(real(w(2:nh))*y2)         Carry along this sum for later use in unfolding the transform.
y2=y2*aimag(w(2:nh))
y(2:nh)=y1-y2                           Calculate the auxiliary function.
y(n:nh+2:-1)=y1+y2
call realft(y(1:n),1)                   Calculate the transform of the auxiliary function.
y(n+1)=y(2)
y(2)=summ                               summ is the value of $F_1$ in equation (12.3.21).
y(2:n:2)=cumsum(y(2:n:2))               Equation (12.3.20).
END SUBROUTINE cosft1
```

```
SUBROUTINE cosft2(y,isign)
USE nrtype; USE nrutil, ONLY : assert,cumsum,zroots_unity
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
INTEGER(I4B), INTENT(IN) :: isign
```

Calculates the "staggered" cosine transform of a set of N real-valued data points y. The transformed data replace the original data in array y. N must be a power of 2. Set isign to $+1$ for a transform, and to -1 for an inverse transform. For an inverse transform, the output array should be multiplied by $2/N$.

```
COMPLEX(SPC), DIMENSION(size(y)) :: w
REAL(SP), DIMENSION(size(y)/2) :: y1,y2
REAL(SP) :: ytemp
INTEGER(I4B) :: n,nh
n=size(y)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in cosft2')
nh=n/2
w=zroots_unity(4*n,n)
if (isign == 1) then                          Forward transform.
    y1=0.5_sp*(y(1:nh)+y(n:nh+1:-1))          Calculate the auxiliary function.
    y2=aimag(w(2:n:2))*(y(1:nh)-y(n:nh+1:-1))
    y(1:nh)=y1+y2
    y(n:nh+1:-1)=y1-y2
    call realft(y,1)                          Calculate transform of the auxiliary function.
    y1(1:nh-1)=y(3:n-1:2)*real(w(3:n-1:2)) &      Even terms.
        -y(4:n:2)*aimag(w(3:n-1:2))
    y2(1:nh-1)=y(4:n:2)*real(w(3:n-1:2)) &
        +y(3:n-1:2)*aimag(w(3:n-1:2))
    y(3:n-1:2)=y1(1:nh-1)
    y(4:n:2)=y2(1:nh-1)
    ytemp=0.5_sp*y(2)                         Initialize recurrence for odd terms with \frac{1}{2}R_{N/2}.
    y(n-2:2:-2)=cumsum(y(n:4:-2),ytemp)           Recurrence for odd terms.
    y(n)=ytemp
else if (isign == -1) then                    Inverse transform.
    ytemp=y(n)
    y(4:n:2)=y(2:n-2:2)-y(4:n:2)              Form difference of odd terms.
    y(2)=2.0_sp*ytemp
    y1(1:nh-1)=y(3:n-1:2)*real(w(3:n-1:2)) &      Calculate R_k and I_k.
        +y(4:n:2)*aimag(w(3:n-1:2))
    y2(1:nh-1)=y(4:n:2)*real(w(3:n-1:2)) &
        -y(3:n-1:2)*aimag(w(3:n-1:2))
    y(3:n-1:2)=y1(1:nh-1)
    y(4:n:2)=y2(1:nh-1)
    call realft(y,-1)
    y1=y(1:nh)+y(n:nh+1:-1)                   Invert auxiliary array.
    y2=(0.5_sp/aimag(w(2:n:2)))*(y(1:nh)-y(n:nh+1:-1))
    y(1:nh)=0.5_sp*(y1+y2)
    y(n:nh+1:-1)=0.5_sp*(y1-y2)
end if
END SUBROUTINE cosft2
```

$\star \quad \star \quad \star$

```
SUBROUTINE four3(data,isign)
USE nrtype
USE nr, ONLY : fourrow_3d
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a 3-d complex array data by its discrete 3-d Fourier transform, if isign is input as 1; or replaces data by its inverse 3-d discrete Fourier transform times the product of its

```
    three sizes, if isign is input as −1. All three of data's sizes must be integer powers of 2
    (this is checked for in fourrow_3d). Parallelism is by use of fourrow_3d.
COMPLEX(SPC), DIMENSION(:,:,:), ALLOCATABLE :: dat2,dat3
call fourrow_3d(data,isign)                    Transform in third dimension.
allocate(dat2(size(data,2),size(data,3),size(data,1)))
dat2=reshape(data,shape=shape(dat2),order=(/3,1,2/))    Transpose.
call fourrow_3d(dat2,isign)                    Transform in (original) first dimension.
allocate(dat3(size(data,3),size(data,1),size(data,2)))
dat3=reshape(dat2,shape=shape(dat3),order=(/3,1,2/))    Transpose.
deallocate(dat2)
call fourrow_3d(dat3,isign)                    Transform in (original) second dimension.
data=reshape(dat3,shape=shape(data),order=(/3,1,2/))    Transpose back to output or-
deallocate(dat3)                                            der.
END SUBROUTINE four3
```

f90 The reshape intrinsic, used with an order= parameter, is the multidimensional generalization of the two-dimensional transpose operation. The line

```
dat2=reshape(data,shape=shape(dat2),order=(/3,1,2/))
```

is equivalent to the do-loop

```
do j=1,size(data,1)
   dat2(:,:,j)=data(j,:,:)
end do
```

Incidentally, we have found some Fortran 90 compilers that (for scalar machines) are significantly *slower* executing the reshape than executing the equivalent do-loop. This, of course, shouldn't happen, since the reshape basically *is* an implicit do-loop. If you find such inefficient behavior on your compiler, you should report it as a bug to your compiler vendor! (Only thus will Fortran 90 compilers be brought to mature states of efficiency.)

```
SUBROUTINE four3_alt(data,isign)
USE nrtype
USE nr, ONLY : fourcol_3d
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
    Replaces a 3-d complex array data by its discrete 2-d Fourier transform, if isign is input
    as 1; or replaces data by its inverse 3-d discrete Fourier transform times the product of
    its three sizes, if isign is input as −1. All three of data's sizes must be integer powers
    of 2 (this is checked for in fourcol_3d). Parallelism is by use of fourcol_3d. (Use this
    version only if fourcol_3d is faster than fourrow_3d on your machine.)
COMPLEX(SPC), DIMENSION(:,:,:), ALLOCATABLE :: dat2,dat3
call fourcol_3d(data,isign)                    Transform in first dimension.
allocate(dat2(size(data,2),size(data,3),size(data,1)))
dat2=reshape(data,shape=shape(dat2),order=(/3,1,2/))    Transpose.
call fourcol_3d(dat2,isign)                    Transform in (original) second dimension.
allocate(dat3(size(data,3),size(data,1),size(data,2)))
dat3=reshape(dat2,shape=shape(dat3),order=(/3,1,2/))    Transpose.
deallocate(dat2)
call fourcol_3d(dat3,isign)                    Transform in (original) third dimension.
data=reshape(dat3,shape=shape(data),order=(/3,1,2/))    Transpose back to output or-
deallocate(dat3)                                            der.
END SUBROUTINE four3_alt
```

Note that four3 uses fourrow_3d, the three-dimensional counterpart of fourrow, while four3_alt uses fourcol_3d, the three-dimensional counterpart of fourcol. You may want to time these programs to see which is faster on your machine.

⋆ ⋆ ⋆

In Volume 1, a single routine named rlft3 was able to serve both as a three-dimensional real FFT, and as a two-dimensional real FFT. The trick is that the Fortran 77 version doesn't care whether the input array data is dimensioned as two- or three-dimensional. Fortran 90 is not so indifferent, and better programming practice is to have two separate versions of the algorithm:

```
SUBROUTINE rlft2(data,spec,speq,isign)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four2
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: spec
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
```

Given a two-dimensional real array data(1:M,1:N), this routine returns (for isign=1) the complex fast Fourier transform as two complex arrays: On output, spec(1:$M/2$,1:N) contains the zero and positive frequency values of the first frequency component, while speq(1:N) contains the Nyquist critical frequency values of the first frequency component. The second frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. For isign=−1, the inverse transform (times $M \times N/2$ as a constant multiplicative factor) is performed, with output data deriving from input spec and speq. For inverse transforms on data not generated first by a forward transform, make sure the complex input data array satisfies property (12.5.2). The size of all arrays must always be integer powers of 2.

```
INTEGER :: i1,j1,nn1,nn2
REAL(DP) :: theta
COMPLEX(SPC) :: c1=(0.5_sp,0.0_sp),c2,h1,h2,w
COMPLEX(SPC), DIMENSION(size(data,2)-1) :: h1a,h2a
COMPLEX(DPC) :: ww,wp
nn1=assert_eq(size(data,1),2*size(spec,1),'rlft2: nn1')
nn2=assert_eq(size(data,2),size(spec,2),size(speq),'rlft2: nn2')
call assert(iand((/nn1,nn2/),(/nn1,nn2/)-1)==0, &
    'dimensions must be powers of 2 in rlft2')
c2=cmplx(0.0_sp,-0.5_sp*isign,kind=spc)
theta=TWOPI_D/(isign*nn1)
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=spc)
if (isign == 1) then                        Case of forward transform.
    spec(:,:)=cmplx(data(1:nn1:2,:),data(2:nn1:2,:),kind=spc)
    call four2(spec,isign)                  Here is where most all of the compute time
    speq=spec(1,:)                              is spent.
end if
h1=c1*(spec(1,1)+conjg(speq(1)))
h1a=c1*(spec(1,2:nn2)+conjg(speq(nn2:2:-1)))
h2=c2*(spec(1,1)-conjg(speq(1)))
h2a=c2*(spec(1,2:nn2)-conjg(speq(nn2:2:-1)))
spec(1,1)=h1+h2
spec(1,2:nn2)=h1a+h2a
speq(1)=conjg(h1-h2)
speq(nn2:2:-1)=conjg(h1a-h2a)
ww=cmplx(1.0_dp,0.0_dp,kind=dpc)            Initialize trigonometric recurrence.
do i1=2,nn1/4+1
    j1=nn1/2-i1+2                           Corresponding negative frequency.
    ww=ww*wp+ww                             Do the trig recurrence.
    w=ww
    h1=c1*(spec(i1,1)+conjg(spec(j1,1)))        Equation (12.3.5).
    h1a=c1*(spec(i1,2:nn2)+conjg(spec(j1,nn2:2:-1)))
```

```
    h2=c2*(spec(i1,1)-conjg(spec(j1,1)))
    h2a=c2*(spec(i1,2:nn2)-conjg(spec(j1,nn2:2:-1)))
    spec(i1,1)=h1+w*h2
    spec(i1,2:nn2)=h1a+w*h2a
    spec(j1,1)=conjg(h1-w*h2)
    spec(j1,nn2:2:-1)=conjg(h1a-w*h2a)
end do
if (isign == -1) then                    Case of reverse transform.
    call four2(spec,isign)
    data(1:nn1:2,:)=real(spec)
    data(2:nn1:2,:)=aimag(spec)
end if
END SUBROUTINE rlft2
```

`call assert(iand((/nn1,nn2/),(/nn1,nn2/)-1)==0 ...` Here an overloaded version of `assert` that takes vector arguments is used to check that each dimension is a power of 2. Note that `iand` acts element-by-element on an array.

```
SUBROUTINE rlft3(data,spec,speq,isign)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : four3
REAL(SP), DIMENSION(:,:,:), INTENT(INOUT) :: data
COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: spec
COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: speq
INTEGER(I4B), INTENT(IN) :: isign
```

Given a three-dimensional real array `data(1:`L`,1:`M`,1:`N`)`, this routine returns (for `isign=1`) the complex Fourier transform as two complex arrays: On output, the zero and positive frequency values of the first frequency component are in `spec(1:`$L/2$`,1:`M`,1:`N`)`, while `speq(1:`M`,1:`N`)` contains the Nyquist critical frequency values of the first frequency component. The second and third frequency components are stored for zero, positive, and negative frequencies, in standard wrap-around order. For `isign=-1`, the inverse transform (times $L \times M \times N/2$ as a constant multiplicative factor) is performed, with output `data` deriving from input `spec` and `speq`. For inverse transforms on data not generated first by a forward transform, make sure the complex input data array satisfies property (12.5.2). The size of all arrays must always be integer powers of 2.

```
INTEGER :: i1,i3,j1,j3,nn1,nn2,nn3
REAL(DP) :: theta
COMPLEX(SPC) :: c1=(0.5_sp,0.0_sp),c2,h1,h2,w
COMPLEX(SPC), DIMENSION(size(data,2)-1) :: h1a,h2a
COMPLEX(DPC) :: ww,wp
c2=cmplx(0.0_sp,-0.5_sp*isign,kind=spc)
nn1=assert_eq(size(data,1),2*size(spec,1),'rlft2: nn1')
nn2=assert_eq(size(data,2),size(spec,2),size(speq,1),'rlft2: nn2')
nn3=assert_eq(size(data,3),size(spec,3),size(speq,2),'rlft2: nn3')
call assert(iand((/nn1,nn2,nn3/),(/nn1,nn2,nn3/)-1)==0, &
    'dimensions must be powers of 2 in rlft3')
theta=TWOPI_D/(isign*nn1)
wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
if (isign == 1) then                     Case of forward transform.
    spec(:,:,:)=cmplx(data(1:nn1:2,:,:),data(2:nn1:2,:,:),kind=spc)
    call four3(spec,isign)               Here is where most all of the compute time
    speq=spec(1,:,:)                         is spent.
end if
do i3=1,nn3
    j3=1
    if (i3 /= 1) j3=nn3-i3+2
    h1=c1*(spec(1,1,i3)+conjg(speq(1,j3)))
    h1a=c1*(spec(1,2:nn2,i3)+conjg(speq(nn2:2:-1,j3)))
    h2=c2*(spec(1,1,i3)-conjg(speq(1,j3)))
    h2a=c2*(spec(1,2:nn2,i3)-conjg(speq(nn2:2:-1,j3)))
    spec(1,1,i3)=h1+h2
    spec(1,2:nn2,i3)=h1a+h2a
```

```
    speq(1,j3)=conjg(h1-h2)
    speq(nn2:2:-1,j3)=conjg(h1a-h2a)
    ww=cmplx(1.0_dp,0.0_dp,kind=dpc)        Initialize trigonometric recurrence.
    do i1=2,nn1/4+1
        j1=nn1/2-i1+2                       Corresponding negative frequency.
        ww=ww*wp+ww                         Do the trig recurrence.
        w=ww
        h1=c1*(spec(i1,1,i3)+conjg(spec(j1,1,j3)))      Equation (12.3.5).
        h1a=c1*(spec(i1,2:nn2,i3)+conjg(spec(j1,nn2:2:-1,j3)))
        h2=c2*(spec(i1,1,i3)-conjg(spec(j1,1,j3)))
        h2a=c2*(spec(i1,2:nn2,i3)-conjg(spec(j1,nn2:2:-1,j3)))
        spec(i1,1,i3)=h1+w*h2
        spec(i1,2:nn2,i3)=h1a+w*h2a
        spec(j1,1,j3)=conjg(h1-w*h2)
        spec(j1,nn2:2:-1,j3)=conjg(h1a-w*h2a)
    end do
end do
if (isign == -1) then                       Case of reverse transform.
    call four3(spec,isign)
    data(1:nn1:2,:,:)=real(spec)
    data(2:nn1:2,:,:)=aimag(spec)
end if
END SUBROUTINE rlft3
```

⋆ ⋆ ⋆

Referring back to the discussion of parallelism, §22.4, that led to `four1`'s implementation with $\sqrt{N}$ parallelism, you might wonder whether Fortran 90 provides sufficiently powerful high-level constructs to enable an FFT routine with N-fold parallelism. The answer is, "*It does*, but you wouldn't want to use them!" Access to arbitrary interprocessor communication in Fortran 90 is through the mechanism of the "vector subscript" (one-dimensional array of indices in arbitrary order). When a vector subscript is on the right-hand side of an assignment statement, the operation performed is effectively a "gather"; when it is on the left-hand side, the operation is effectively a "scatter."

It is quite possible to write the classic FFT algorithm in terms of gather and scatter operations. In fact, we do so now. The problem is efficiency: The computations involved in constructing the vector subscripts for the scatter/gather operations, and the actual scatter/gather operations themselves, tend to swamp the underlying very lean FFT algorithm. The result is very slow, though theoretically perfectly parallelizable, code. Since small-scale parallel (SSP) machines can saturate their processors with $\sqrt{N}$ parallelism, while massively multiprocessor (MMP) machines inevitably come with architecture-optimized FFT library calls, there is really no niche for these routines, except as pedagogical demonstrations. We give here a one-dimensional routine, and also an arbitrary-dimensional routine modeled on Volume 1's `fourn`. Note the complete absence of do-loops of size N; the loops that remain are over $\log N$ stages, or over the number of dimensions.

```
SUBROUTINE four1_gather(data,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), INTENT(IN) :: isign
```

Replaces a complex array `data` by its discrete Fourier transform, if `isign` is input as 1; or replaces `data` by `size(data)` times its inverse discrete Fourier transform, if `isign` is input as -1. The size of `data` must be an integer power of 2. This routine demonstrates coding the FFT algorithm in high-level Fortran 90 constructs. Generally the result is *very*

much slower than library routines coded for specific architectures, and also *significantly slower* than the parallelization-by-rows method used in the routine four1.

```
INTEGER(I4B) :: n,n2,m,mm
REAL(DP) :: theta
COMPLEX(SPC) :: wp
INTEGER(I4B), DIMENSION(size(data)) :: jarr
INTEGER(I4B), DIMENSION(:), ALLOCATABLE :: jrev
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: wtab,dtemp
n=size(data)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in four1_gather')
if (n <= 1) RETURN
allocate(jrev(n))                          Begin bit-reversal section of the routine.
jarr=arth(0,1,n)
jrev=0
n2=n/2
m=n2
do                                         Construct an array of pointers from an index
    where (iand(jarr,1) /= 0) jrev=jrev+m        to its bit-reverse.
    jarr=jarr/2
    m=m/2
    if (m == 0) exit
end do
data=data(jrev+1)                          Move all data to bit-reversed location by a
deallocate(jrev)                               single gather/scatter.
allocate(dtemp(n),wtab(n2))                Begin Danielson-Lanczos section of the rou-
jarr=arth(0,1,n)                               tine.
m=1
mm=n2
wtab(1)=(1.0_sp,0.0_sp)                    Seed the roots-of-unity table.
do                                         Outer loop executed log2 N times.
    where (iand(jarr,m) /= 0)
          The basic idea is to address the correct root-of-unity for each Danielson-Lanczos
          multiplication by tricky bit manipulations.
        dtemp=data*wtab(mm*iand(jarr,m-1)+1)
        data=eoshift(data,-m)-dtemp        This is half of Danielson-Lanczos.
    elsewhere
        data=data+eoshift(dtemp,m)         This is the other half. The referenced ele-
    end where                                  ments of dtemp will have been set in the
    m=m*2                                      where clause.
    if (m >= n) exit
    mm=mm/2
    theta=PI_D/(isign*m)                   Ready for trigonometry?
    wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2, sin(theta),kind=spc)
      Add entries to the table for the next iteration.
    wtab(mm+1:n2:2*mm)=wtab(1:n2-mm:2*mm)*wp+wtab(1:n2-mm:2*mm)
end do
deallocate(dtemp,wtab)
END SUBROUTINE four1_gather
```

```
SUBROUTINE fourn_gather(data,nn,isign)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
INTEGER(I4B), DIMENSION(:) :: nn
INTEGER(I4B), INTENT(IN) :: isign
```

For data a one-dimensional complex array containing the values (in Fortran normal ordering) of an M-dimensional complex arrray, this routine replaces data by its M-dimensional discrete Fourier transform, if isign is input as 1. nn(1:M) is an integer array containing the lengths of each dimension (number of complex values), each of which must be a power of 2. If isign is input as -1, data is replaced by its inverse transform times the product of the lengths of all dimensions. This routine demonstrates coding the multidimensional FFT algorithm in high-level Fortran 90 constructs. Generally the result is *very much slower* than

library routines coded for specific architectures, and *significantly slower* than routines four2 and four3 for the two- and three-dimensional cases.

```
INTEGER(I4B), DIMENSION(:), ALLOCATABLE :: jarr
INTEGER(I4B) :: ndim,idim,ntot,nprev,n,n2,msk0,msk1,msk2,m,mm,mn
REAL(DP) :: theta
COMPLEX(SPC) :: wp
COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: wtab,dtemp
call assert(iand(nn,nn-1)==0, &
    'each dimension must be a power of 2 in fourn_gather')
ndim=size(nn)
ntot=product(nn)
nprev=1
allocate(jarr(ntot))
do idim=1,ndim                                  Loop over the dimensions.
    jarr=arth(0,1,ntot)                         We begin the bit-reversal section of the
    n=nn(idim)                                      routine.
    n2=n/2
    msk0=nprev
    msk1=nprev*n2
    msk2=msk0+msk1
    do                                          Construct an array of pointers from an
        if (msk1 <= msk0) exit                      index to its bit-reverse.
        where (iand(jarr,msk0) == 0 .neqv. iand(jarr,msk1) == 0) &
            jarr=ieor(jarr,msk2)
        msk0=msk0*2
        msk1=msk1/2
        msk2=msk0+msk1
    end do
    data=data(jarr+1)                           Move all data to bit-reversed location by
    allocate(dtemp(ntot),wtab(n2))                  a single gather/scatter.
      We begin the Danielson-Lanczos section of the routine.
    jarr=iand(n-1,arth(0,1,ntot)/nprev)
    m=1
    mm=n2
    mn=m*nprev
    wtab(1)=(1.0_sp,0.0_sp)                     Seed the roots-of-unity table.
    do                                          This loop executed log2 N times.
        if (mm == 0) exit
        where (iand(jarr,m) /= 0)
              The basic idea is to address the correct root-of-unity for each Danielson-Lanczos
              multiplication by tricky bit manipulations.
            dtemp=data*wtab(mm*iand(jarr,m-1)+1)
            data=eoshift(data,-mn)-dtemp        This is half of Danielson-Lanczos.
        elsewhere
            data=data+eoshift(dtemp,mn)         This is the other half. The referenced el-
        end where                                   ements of dtemp will have been set
        m=m*2                                       in the where clause.
        if (m >= n) exit
        mn=m*nprev
        mm=mm/2
        theta=PI_D/(isign*m)                    Ready for trigonometry?
        wp=cmplx(-2.0_dp*sin(0.5_dp*theta)**2,sin(theta),kind=dpc)
          Add entries to the table for the next iteration.
        wtab(mm+1:n2:2*mm)=wtab(1:n2-mm:2*mm)*wp &
            +wtab(1:n2-mm:2*mm)
    end do
    deallocate(dtemp,wtab)
    nprev=n*nprev
end do
deallocate(jarr)
END SUBROUTINE fourn_gather
```

f90 `call assert(iand(nn,nn-1)==0 ...` Once again the vector version of `assert` is used to test all the dimensions stored in `nn` simultaneously.

Chapter B13. Fourier and Spectral Applications

```
FUNCTION convlv(data,respns,isign)
USE nrtype; USE nrutil, ONLY : assert,nrerror
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
REAL(SP), DIMENSION(:), INTENT(IN) :: respns
INTEGER(I4B), INTENT(IN) :: isign
REAL(SP), DIMENSION(size(data)) :: convlv
```

Convolves or deconvolves a real data set `data` (of length N, including any user-supplied zero padding) with a response function `respns`, stored in wrap-around order in a real array of length $M \leq N$. (M should be an odd integer, N a power of 2.) Wrap-around order means that the first half of the array `respns` contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element `respns`(M). On input `isign` is $+1$ for convolution, -1 for deconvolution. The answer is returned as the function `convlv`, an array of length N. `data` has `INTENT(INOUT)` for consistency with `realft`, but is actually unchanged.

```
INTEGER(I4B) :: no2,n,m
COMPLEX(SPC), DIMENSION(size(data)/2) :: tmpd,tmpr
n=size(data)
m=size(respns)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in convlv')
call assert(mod(m,2)==1, 'm must be odd in convlv')
convlv(1:m)=respns(:)                   Put respns in array of length n.
convlv(n-(m-3)/2:n)=convlv((m+3)/2:m)
convlv((m+3)/2:n-(m-1)/2)=0.0           Pad with zeros.
no2=n/2
call realft(data,1,tmpd)                FFT both arrays.
call realft(convlv,1,tmpr)
if (isign == 1) then                    Multiply FFTs to convolve.
    tmpr(1)=cmplx(real(tmpd(1))*real(tmpr(1))/no2, &
        aimag(tmpd(1))*aimag(tmpr(1))/no2, kind=spc)
    tmpr(2:)=tmpd(2:)*tmpr(2:)/no2
else if (isign == -1) then              Divide FFTs to deconvolve.
    if (any(abs(tmpr(2:)) == 0.0) .or. real(tmpr(1)) == 0.0 &
        .or. aimag(tmpr(1)) == 0.0) call nrerror &
        ('deconvolving at response zero in convlv')
    tmpr(1)=cmplx(real(tmpd(1))/real(tmpr(1))/no2, &
        aimag(tmpd(1))/aimag(tmpr(1))/no2, kind=spc)
    tmpr(2:)=tmpd(2:)/tmpr(2:)/no2
else
    call nrerror('no meaning for isign in convlv')
end if
call realft(convlv,-1,tmpr)             Inverse transform back to time domain.
END FUNCTION convlv
```

f90 tmpr(1)=cmplx(...kind=spc) The intrinsic function cmplx returns a quantity of type default complex unless the kind argument is present. It is therefore a good idea always to include this argument. The intrinsic functions real and aimag, on the other hand, when called with a complex argument, return the same kind as their argument. So it is a good idea *not* to put in a kind argment for these. (In fact, aimag doesn't allow one.) Don't confuse these situations, regarding complex variables, with the completely unrelated use of real to convert a real or integer variable to a real value of specified kind. In this latter case, kind should be specified.

⋆ ⋆ ⋆

```
FUNCTION correl(data1,data2)
USE nrtype; USE nrutil, ONLY : assert,assert_eq
USE nr, ONLY : realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data1,data2
REAL(SP), DIMENSION(size(data1)) :: correl
```

Computes the correlation of two real data sets data1 and data2 of length N (including any user-supplied zero padding). N must be an integer power of 2. The answer is returned as the function correl, an array of length N. The answer is stored in wrap-around order, i.e., correlations at increasingly negative lags are in correl(N) on down to correl($N/2+1$), while correlations at increasingly positive lags are in correl(1) (zero lag) on up to correl($N/2$). Sign convention of this routine: if data1 lags data2, i.e., is shifted to the right of it, then correl will show a peak at positive lags.

```
COMPLEX(SPC), DIMENSION(size(data1)/2) :: cdat1,cdat2
INTEGER(I4B) :: no2,n                          Normalization for inverse FFT.
n=assert_eq(size(data1),size(data2),'correl')
call assert(iand(n,n-1)==0, 'n must be a power of 2 in correl')
no2=n/2
call realft(data1,1,cdat1)                     Transform both data vectors.
call realft(data2,1,cdat2)
cdat1(1)=cmplx(real(cdat1(1))*real(cdat2(1))/no2, &     Multiply to find FFT of their
    aimag(cdat1(1))*aimag(cdat2(1))/no2, kind=spc)          correlation.
cdat1(2:)=cdat1(2:)*conjg(cdat2(2:))/no2
call realft(correl,-1,cdat1)                   Inverse transform gives correlation.
END FUNCTION correl
```

f90 cdat1(1)=cmplx(...kind=spc) See just above for why we use the explicit kind type parameter spc for cmplx, but omit sp for real.

⋆ ⋆ ⋆

```
SUBROUTINE spctrm(p,k,ovrlap,unit,n_window)
USE nrtype; USE nrutil, ONLY : arth,nrerror
USE nr, ONLY : four1
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(OUT) :: p
INTEGER(I4B), INTENT(IN) :: k
LOGICAL(LGT), INTENT(IN) :: ovrlap             True for overlapping segments, false other-
INTEGER(I4B), OPTIONAL, INTENT(IN) :: n_window,unit     wise.
```

Reads data from input unit 9, or if the optional argument unit is present, from that input unit. The output is an array p of length M that contains the data's power (mean square amplitude) at frequency $(j-1)/2M$ cycles per grid point, for $j = 1, 2, \ldots, M$, based on (2*k+1)*M data points (if ovrlap is set .true.) or 4*k*M data points (if ovrlap is set .false.). The number of segments of the data is 2*k in both cases: The routine calls four1 k times, each call with 2 partitions each of $2M$ real data points. If the optional argument n_window is present, the routine uses the Bartlett window, the square window,

or the Welch window for n_window $= 1, 2, 3$ respectively. If n_window is not present, the Bartlett window is used.

```
INTEGER(I4B) :: j,joff,joffn,kk,m,m4,m43,m44,mm,iunit,nn_window
REAL(SP) :: den,facm,facp,sumw
REAL(SP), DIMENSION(2*size(p)) :: w
REAL(SP), DIMENSION(4*size(p)) :: w1
REAL(SP), DIMENSION(size(p)) :: w2
COMPLEX(SPC), DIMENSION(2*size(p)) :: cw1
m=size(p)
if (present(n_window)) then
    nn_window=n_window
else
    nn_window=1
end if
if (present(unit)) then
    iunit=unit
else
    iunit=9
end if
mm=m+m                                  Useful factors.
m4=mm+mm
m44=m4+4
m43=m4+3
den=0.0
facm=m                                  Factors used by the window function.
facp=1.0_sp/m
w1(1:mm)=window(arth(1,1,mm),facm,facp,nn_window)
sumw=dot_product(w1(1:mm),w1(1:mm))     Accumulate the squared sum of the weights.
p(:)=0.0                                Initialize the spectrum to zero.
if (ovrlap) read (iunit,*) (w2(j),j=1,m)    Initialize the "save" half-buffer.
do kk=1,k                               Loop over data segments in groups of two.
    do joff=-1,0,1                      Get two complete segments into workspace.
        if (ovrlap) then
            w1(joff+2:joff+mm:2)=w2(1:m)
            read (iunit,*) (w2(j),j=1,m)
            joffn=joff+mm
            w1(joffn+2:joffn+mm:2)=w2(1:m)
        else
            read (iunit,*) (w1(j),j=joff+2,m4,2)
        end if
    end do
    w=window(arth(1,1,mm),facm,facp,nn_window)      Apply the window to the data.
    w1(2:m4:2)=w1(2:m4:2)*w
    w1(1:m4:2)=w1(1:m4:2)*w
    cw1(1:mm)=cmplx(w1(1:m4:2),w1(2:m4:2),kind=spc)
    call four1(cw1(1:mm),1)             Fourier transform the windowed data.
    w1(1:m4:2)=real(cw1(1:mm))
    w1(2:m4:2)=aimag(cw1(1:mm))
    p(1)=p(1)+w1(1)**2+w1(2)**2         Sum results into previous segments.
    p(2:m)=p(2:m)+w1(4:2*m:2)**2+w1(3:2*m-1:2)**2+&
        w1(m44-4:m44-2*m:-2)**2+w1(m43-4:m43-2*m:-2)**2
    den=den+sumw
end do
p(:)=p(:)/(m4*den)                      Normalize the output.
CONTAINS

FUNCTION window(j,facm,facp,nn_window)
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: j
INTEGER(I4B), INTENT(IN) :: nn_window
REAL(SP), INTENT(IN) :: facm,facp
REAL(SP), DIMENSION(size(j)) :: window
select case(nn_window)
    case(1)
```

```
        window(j)=(1.0_sp-abs(((j-1)-facm)*facp))        Bartlett window.
    case(2)
        window(j)=1.0                                    Square window.
    case(3)
        window(j)=(1.0_sp-(((j-1)-facm)*facp)**2)        Welch window.
    case default
        call nrerror('unimplemented window function in spctrm')
end select
END FUNCTION window
END SUBROUTINE spctrm
```

The Fortran 90 optional argument feature allows us to make unit 9 the default output unit in this routine, but leave the user the option of specifying a different output unit by supplying an actual argument for `unit`. We also use an optional argument to allow the user the option of overriding the default selection of the Bartlett window function.

`FUNCTION window(j,facm,facp,nn_window)` In Fortran 77 we coded this as a statement function. Here the internal function is equivalent, but allows full specification of the interface and so is preferred.

⋆ ⋆ ⋆

```
SUBROUTINE memcof(data,xms,d)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: xms
REAL(SP), DIMENSION(:), INTENT(IN) :: data
REAL(SP), DIMENSION(:), INTENT(OUT) :: d
```

Given a real vector `data` of length N, this routine returns M linear prediction coefficients in a vector `d` of length M, and returns the mean square discrepancy as `xms`.

```
INTEGER(I4B) :: k,m,n
REAL(SP) :: denom,pneum
REAL(SP), DIMENSION(size(data)) :: wk1,wk2,wktmp
REAL(SP), DIMENSION(size(d)) :: wkm
m=size(d)
n=size(data)
xms=dot_product(data,data)/n
wk1(1:n-1)=data(1:n-1)
wk2(1:n-1)=data(2:n)
do k=1,m
    pneum=dot_product(wk1(1:n-k),wk2(1:n-k))
    denom=dot_product(wk1(1:n-k),wk1(1:n-k))+ &
        dot_product(wk2(1:n-k),wk2(1:n-k))
    d(k)=2.0_sp*pneum/denom
    xms=xms*(1.0_sp-d(k)**2)
    d(1:k-1)=wkm(1:k-1)-d(k)*wkm(k-1:1:-1)
```

The algorithm is recursive, although it is implemented as an iteration. It builds up the answer for larger and larger values of `m` until the desired value is reached. At this point in the algorithm, one could return the vector `d` and scalar `xms` for a set of LP coefficients with `k` (rather than `m`) terms.

```
    if (k == m) RETURN
    wkm(1:k)=d(1:k)
    wktmp(2:n-k)=wk1(2:n-k)
    wk1(1:n-k-1)=wk1(1:n-k-1)-wkm(k)*wk2(1:n-k-1)
    wk2(1:n-k-1)=wk2(2:n-k)-wkm(k)*wktmp(2:n-k)
end do
call nrerror('never get here in memcof')
END SUBROUTINE memcof
```

⋆ ⋆ ⋆

```
SUBROUTINE fixrts(d)
USE nrtype
USE nr, ONLY : zroots
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
```
Given the LP coefficients d, this routine finds all roots of the characteristic polynomial (13.6.14), reflects any roots that are outside the unit circle back inside, and then returns a modified set of coefficients in d.
```
INTEGER(I4B) :: i,m
LOGICAL(LGT) :: polish
COMPLEX(SPC), DIMENSION(size(d)+1) :: a
COMPLEX(SPC), DIMENSION(size(d)) :: roots
m=size(d)
a(m+1)=cmplx(1.0_sp,kind=spc)              Set up complex coefficients for polynomial
a(m:1:-1)=cmplx(-d(1:m),kind=spc)             root finder.
polish=.true.
call zroots(a(1:m+1),roots,polish)         Find all the roots.
where (abs(roots) > 1.0) roots=1.0_sp/conjg(roots)
```
Reflect all roots outside the unit circle back inside.
```
a(1)=-roots(1)                             Now reconstruct the polynomial coefficients,
a(2:m+1)=cmplx(1.0_sp,kind=spc)
do i=2,m                                   by looping over the roots
    a(2:i)=a(1:i-1)-roots(i)*a(2:i)        and synthetically multiplying.
    a(1)=-roots(i)*a(1)
end do
d(m:1:-1)=-real(a(1:m))                    The polynomial coefficients are guaranteed
END SUBROUTINE fixrts                         to be real, so we need only return the
                                              real part as new LP coefficients.
```

f90 `a(m+1)=cmplx(1.0_sp,kind=spc)` See after `convlv` on p. 1254 to review why we use the explicit kind type parameter `spc` for `cmplx`.

⋆ ⋆ ⋆

```
FUNCTION predic(data,d,nfut)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data,d
INTEGER(I4B), INTENT(IN) :: nfut
REAL(SP), DIMENSION(nfut) :: predic
```
Given an array data, and given the data's LP coefficients d in an array of length M, this routine applies equation (13.6.11) to predict the next nfut data points, which it returns in an array as the function value predic. Note that the routine references only the last M values of data, as initial values for the prediction.
```
INTEGER(I4B) :: j,ndata,m
REAL(SP) :: discrp,sm
REAL(SP), DIMENSION(size(d)) :: reg
m=size(d)
ndata=size(data)
reg(1:m)=data(ndata:ndata+1-m:-1)
do j=1,nfut
    discrp=0.0
```
This is where you would put in a known discrepancy if you were reconstructing a function by linear predictive coding rather than extrapolating a function by linear prediction. See text.
```
    sm=discrp+dot_product(d,reg)
    reg=eoshift(reg,-1,sm)          [If you want to implement circular arrays, you can
    predic(j)=sm                       avoid this shifting of coefficients!]
end do
END FUNCTION predic
```

⋆ ⋆ ⋆

```
FUNCTION evlmem(fdt,d,xms)
USE nrtype; USE nrutil, ONLY : poly
IMPLICIT NONE
REAL(SP), INTENT(IN) :: fdt,xms
REAL(SP), DIMENSION(:), INTENT(IN) :: d
REAL(SP) :: evlmem
```

Given `d` and `xms` as returned by `memcof`, this function returns the power spectrum estimate $P(f)$ as a function of `fdt` $= f\Delta$.

```
COMPLEX(SPC) :: z,zz
REAL(DP) :: theta                Trigonometric recurrences in double precision.
theta=TWOPI_D*fdt
z=cmplx(cos(theta),sin(theta),kind=spc)
zz=1.0_sp-z*poly(z,d)
evlmem=xms/abs(zz)**2            Equation (13.7.4).
END FUNCTION evlmem
```

f90 `zz=...poly(z,d)` The `poly` function in `nrutil` returns the value of the polynomial with coefficients `d(:)` at `z`. Here a version that takes real coefficients and a complex argument is actually invoked, but all the different versions have been overloaded onto the same name `poly`.

⋆ ⋆ ⋆

```
SUBROUTINE period(x,y,ofac,hifac,px,py,jmax,prob)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc
USE nr, ONLY : avevar
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: jmax
REAL(SP), INTENT(IN) :: ofac,hifac
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:), POINTER :: px,py
```

Input is a set of N data points with abscissas `x` (which need not be equally spaced) and ordinates `y`, and a desired oversampling factor `ofac` (a typical value being 4 or larger). The routine returns pointers to internally allocated arrays `px` and `py`. `px` is filled with an increasing sequence of frequencies (not angular frequencies) up to `hifac` times the "average" Nyquist frequency, and `py` is filled with the values of the Lomb normalized periodogram at those frequencies. The length of these arrays is `0.5*ofac*hifac*`N. The arrays `x` and `y` are not altered. The routine also returns `jmax` such that `py(jmax)` is the maximum element in `py`, and `prob`, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of `prob` indicates that a significant periodic signal is present.

```
INTEGER(I4B) :: i,n,nout
REAL(SP) :: ave,cwtau,effm,expy,pnow,sumc,sumcy,&
    sums,sumsh,sumsy,swtau,var,wtau,xave,xdif,xmax,xmin
REAL(DP), DIMENSION(size(x)) :: tmp1,tmp2,wi,wpi,wpr,wr
LOGICAL(LGT), SAVE :: init=.true.
n=assert_eq(size(x),size(y),'period')
if (init) then
    init=.false.
    nullify(px,py)
else
    if (associated(px)) deallocate(px)
    if (associated(py)) deallocate(py)
end if
nout=0.5_sp*ofac*hifac*n
allocate(px(nout),py(nout))
call avevar(y(:),ave,var)                Get mean and variance of the input data.
xmax=maxval(x(:))                        Go through data to get the range of abscis-
xmin=minval(x(:))                            sas.
xdif=xmax-xmin
```

```
xave=0.5_sp*(xmax+xmin)
pnow=1.0_sp/(xdif*ofac)                    Starting frequency.
tmp1(:)=TWOPI_D*((x(:)-xave)*pnow)         Initialize values for the trigonometric recur-
wpr(:)=-2.0_dp*sin(0.5_dp*tmp1)**2             rences at each data point. The recur-
wpi(:)=sin(tmp1(:))                            rences are done in double precision.
wr(:)=cos(tmp1(:))
wi(:)=wpi(:)
do i=1,nout                                Main loop over the frequencies to be evalu-
    px(i)=pnow                                 ated.
    sumsh=dot_product(wi,wr)               First, loop over the data to get τ and related
    sumc=dot_product(wr(:)-wi(:),wr(:)+wi(:))      quantities.
    wtau=0.5_sp*atan2(2.0_sp*sumsh,sumc)
    swtau=sin(wtau)
    cwtau=cos(wtau)
    tmp1(:)=wi(:)*cwtau-wr(:)*swtau        Then, loop over the data again to get the
    tmp2(:)=wr(:)*cwtau+wi(:)*swtau            periodogram value.
    sums=dot_product(tmp1,tmp1)
    sumc=dot_product(tmp2,tmp2)
    sumsy=dot_product(y(:)-ave,tmp1)
    sumcy=dot_product(y(:)-ave,tmp2)
    tmp1(:)=wr(:)                          Update the trigonometric recurrences.
    wr(:)=(wr(:)*wpr(:)-wi(:)*wpi(:))+wr(:)
    wi(:)=(wi(:)*wpr(:)+tmp1(:)*wpi(:))+wi(:)
    py(i)=0.5_sp*(sumcy**2/sumc+sumsy**2/sums)/var
    pnow=pnow+1.0_sp/(ofac*xdif)           The next frequency.
end do
jmax=imaxloc(py(1:nout))
expy=exp(-py(jmax))                        Evaluate statistical significance of the maxi-
effm=2.0_sp*nout/ofac                          mum.
prob=effm*expy
if (prob > 0.01_sp) prob=1.0_sp-(1.0_sp-expy)**effm
END SUBROUTINE period
```

f90 This routine shows another example of how to return arrays whose size is not known in advance (cf. zbrac in Chapter B9). The coding is explained in the subsection on pointers in §21.5. The size of the output arrays, nout in the code, is available as size(px).

jmax=imaxloc... See discussion of imaxloc on p. 1017.

```
SUBROUTINE fasper(x,y,ofac,hifac,px,py,jmax,prob)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,imaxloc,nrerror
USE nr, ONLY : avevar,realft
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(IN) :: ofac,hifac
INTEGER(I4B), INTENT(OUT) :: jmax
REAL(SP), INTENT(OUT) :: prob
REAL(SP), DIMENSION(:), POINTER :: px,py
INTEGER(I4B), PARAMETER :: MACC=4
```

Input is a set of N data points with abscissas x (which need not be equally spaced) and ordinates y, and a desired oversampling factor ofac (a typical value being 4 or larger). The routine returns pointers to internally allocated arrays px and py. px is filled with an increasing sequence of frequencies (not angular frequencies) up to hifac times the "average" Nyquist frequency, and py is filled with the values of the Lomb normalized periodogram at those frequencies. The length of these arrays is 0.5*ofac*hifac*N. The arrays x and y are not altered. The routine also returns jmax such that py(jmax) is the maximum element in py, and prob, an estimate of the significance of that maximum against the hypothesis of random noise. A small value of prob indicates that a significant

periodic signal is present.
Parameter: MACC is the number of interpolation points per 1/4 cycle of highest frequency.

```
INTEGER(I4B) :: j,k,n,ndim,nfreq,nfreqt,nout
REAL(SP) :: ave,ck,ckk,cterm,cwt,den,df,effm,expy,fac,fndim,hc2wt,&
    hs2wt,hypo,sterm,swt,var,xdif,xmax,xmin
REAL(SP), DIMENSION(:), ALLOCATABLE :: wk1,wk2
LOGICAL(LGT), SAVE :: init=.true.
n=assert_eq(size(x),size(y),'fasper')
if (init) then
    init=.false.
    nullify(px,py)
else
    if (associated(px)) deallocate(px)
    if (associated(py)) deallocate(py)
end if
nfreqt=ofac*hifac*n*MACC
nfreq=64
do                                  Size the FFT as next power of 2 above nfreqt.
    if (nfreq >= nfreqt) exit
    nfreq=nfreq*2
end do
ndim=2*nfreq
allocate(wk1(ndim),wk2(ndim))
call avevar(y(1:n),ave,var)         Compute the mean, variance, and range of the data.
xmax=maxval(x(:))
xmin=minval(x(:))
xdif=xmax-xmin
wk1(1:ndim)=0.0                     Zero the workspaces.
wk2(1:ndim)=0.0
fac=ndim/(xdif*ofac)
fndim=ndim
do j=1,n                            Extirpolate the data into the workspaces.
    ck=1.0_sp+mod((x(j)-xmin)*fac,fndim)
    ckk=1.0_sp+mod(2.0_sp*(ck-1.0_sp),fndim)
    call spreadval(y(j)-ave,wk1,ck,MACC)
    call spreadval(1.0_sp,wk2,ckk,MACC)
end do
call realft(wk1(1:ndim),1)          Take the fast Fourier transforms.
call realft(wk2(1:ndim),1)
df=1.0_sp/(xdif*ofac)
nout=0.5_sp*ofac*hifac*n
allocate(px(nout),py(nout))
k=3
do j=1,nout                         Compute the Lomb value for each frequency.
    hypo=sqrt(wk2(k)**2+wk2(k+1)**2)
    hc2wt=0.5_sp*wk2(k)/hypo
    hs2wt=0.5_sp*wk2(k+1)/hypo
    cwt=sqrt(0.5_sp+hc2wt)
    swt=sign(sqrt(0.5_sp-hc2wt),hs2wt)
    den=0.5_sp*n+hc2wt*wk2(k)+hs2wt*wk2(k+1)
    cterm=(cwt*wk1(k)+swt*wk1(k+1))**2/den
    sterm=(cwt*wk1(k+1)-swt*wk1(k))**2/(n-den)
    px(j)=j*df
    py(j)=(cterm+sterm)/(2.0_sp*var)
    k=k+2
end do
deallocate(wk1,wk2)
jmax=imaxloc(py(1:nout))
expy=exp(-py(jmax))                 Estimate significance of largest peak value.
effm=2.0_sp*nout/ofac
prob=effm*expy
if (prob > 0.01_sp) prob=1.0_sp-(1.0_sp-expy)**effm
CONTAINS
```

```
SUBROUTINE spreadval(y,yy,x,m)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: y,x
REAL(SP), DIMENSION(:), INTENT(INOUT) :: yy
INTEGER(I4B), INTENT(IN) :: m
```

Given an array yy of length N, extirpolate (spread) a value y into m actual array elements that best approximate the "fictional" (i.e., possibly noninteger) array element number x. The weights used are coefficients of the Lagrange interpolating polynomial.

```
INTEGER(I4B) :: ihi,ilo,ix,j,nden,n
REAL(SP) :: fac
INTEGER(I4B), DIMENSION(10) :: nfac = (/ &
    1,1,2,6,24,120,720,5040,40320,362880 /)
if (m > 10) call nrerror('factorial table too small in spreadval')
n=size(yy)
ix=x
if (x == real(ix,sp)) then
    yy(ix)=yy(ix)+y
else
    ilo=min(max(int(x-0.5_sp*m+1.0_sp),1),n-m+1)
    ihi=ilo+m-1
    nden=nfac(m)
    fac=product(x-arth(ilo,1,m))
    yy(ihi)=yy(ihi)+y*fac/(nden*(x-ihi))
    do j=ihi-1,ilo,-1
        nden=(nden/(j+1-ilo))*(j-ihi)
        yy(j)=yy(j)+y*fac/(nden*(x-j))
    end do
end if
END SUBROUTINE spreadval
END SUBROUTINE fasper
```

f90 This routine shows another example of how to return arrays whose size is not known in advance (cf. zbrac in Chapter B9). The coding is explained in the subsection on pointers in §21.5. The size of the output arrays, nout in the code, is available as size(px).

`jmax=imaxloc...` See discussion of imaxloc on p. 1017.

`if (x == real(ix,sp)) then` Without the explicit kind type parameter sp, real returns a value of type default real for an integer argument. This prevents automatic conversion of the routine from single to double precision. Here all you have to do is redefine sp in nrtype to get double precision.

⋆ ⋆ ⋆

```
SUBROUTINE dftcor(w,delta,a,b,endpts,corre,corim,corfac)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), INTENT(IN) :: w,delta,a,b
REAL(SP), INTENT(OUT) :: corre,corim,corfac
REAL(SP), DIMENSION(:), INTENT(IN) :: endpts
```

For an integral approximated by a discrete Fourier transform, this routine computes the correction factor that multiplies the DFT and the endpoint correction to be added. Input is the angular frequency w, stepsize delta, lower and upper limits of the integral a and b, while the array endpts of length 8 contains the first 4 and last 4 function values. The

correction factor $W(\theta)$ is returned as corfac, while the real and imaginary parts of the endpoint correction are returned as corre and corim.

```
REAL(SP) :: a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c,cl,cr,s,sl,sr,t,&
    t2,t4,t6
REAL(DP) :: cth,ctth,spth2,sth,sth4i,stth,th,th2,th4,&
    tmth2,tth4i
th=w*delta
call assert(a < b, th >= 0.0, th <= PI_D, 'dftcor args')
if (abs(th) < 5.0e-2_dp) then          Use series.
    t=th
    t2=t*t
    t4=t2*t2
    t6=t4*t2
    corfac=1.0_sp-(11.0_sp/720.0_sp)*t4+(23.0_sp/15120.0_sp)*t6
    a0r=(-2.0_sp/3.0_sp)+t2/45.0_sp+(103.0_sp/15120.0_sp)*t4-&
        (169.0_sp/226800.0_sp)*t6
    a1r=(7.0_sp/24.0_sp)-(7.0_sp/180.0_sp)*t2+(5.0_sp/3456.0_sp)*t4&
        -(7.0_sp/259200.0_sp)*t6
    a2r=(-1.0_sp/6.0_sp)+t2/45.0_sp-(5.0_sp/6048.0_sp)*t4+t6/64800.0_sp
    a3r=(1.0_sp/24.0_sp)-t2/180.0_sp+(5.0_sp/24192.0_sp)*t4-t6/259200.0_sp
    a0i=t*(2.0_sp/45.0_sp+(2.0_sp/105.0_sp)*t2-&
        (8.0_sp/2835.0_sp)*t4+(86.0_sp/467775.0_sp)*t6)
    a1i=t*(7.0_sp/72.0_sp-t2/168.0_sp+(11.0_sp/72576.0_sp)*t4-&
        (13.0_sp/5987520.0_sp)*t6)
    a2i=t*(-7.0_sp/90.0_sp+t2/210.0_sp-(11.0_sp/90720.0_sp)*t4+&
        (13.0_sp/7484400.0_sp)*t6)
    a3i=t*(7.0_sp/360.0_sp-t2/840.0_sp+(11.0_sp/362880.0_sp)*t4-&
        (13.0_sp/29937600.0_sp)*t6)
else                                   Use trigonometric formulas in double precision.
    cth=cos(th)
    sth=sin(th)
    ctth=cth**2-sth**2
    stth=2.0_dp*sth*cth
    th2=th*th
    th4=th2*th2
    tmth2=3.0_dp-th2
    spth2=6.0_dp+th2
    sth4i=1.0_sp/(6.0_dp*th4)
    tth4i=2.0_dp*sth4i
    corfac=tth4i*spth2*(3.0_sp-4.0_dp*cth+ctth)
    a0r=sth4i*(-42.0_dp+5.0_dp*th2+spth2*(8.0_dp*cth-ctth))
    a0i=sth4i*(th*(-12.0_dp+6.0_dp*th2)+spth2*stth)
    a1r=sth4i*(14.0_dp*tmth2-7.0_dp*spth2*cth)
    a1i=sth4i*(30.0_dp*th-5.0_dp*spth2*sth)
    a2r=tth4i*(-4.0_dp*tmth2+2.0_dp*spth2*cth)
    a2i=tth4i*(-12.0_dp*th+2.0_dp*spth2*sth)
    a3r=sth4i*(2.0_dp*tmth2-spth2*cth)
    a3i=sth4i*(6.0_dp*th-spth2*sth)
end if
cl=a0r*endpts(1)+a1r*endpts(2)+a2r*endpts(3)+a3r*endpts(4)
sl=a0i*endpts(1)+a1i*endpts(2)+a2i*endpts(3)+a3i*endpts(4)
cr=a0r*endpts(8)+a1r*endpts(7)+a2r*endpts(6)+a3r*endpts(5)
sr=-a0i*endpts(8)-a1i*endpts(7)-a2i*endpts(6)-a3i*endpts(5)
arg=w*(b-a)
c=cos(arg)
s=sin(arg)
corre=cl+c*cr-s*sr
corim=sl+s*cr+c*sr
END SUBROUTINE dftcor
```

```
SUBROUTINE dftint(func,a,b,w,cosint,sinint)
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : dftcor,polint,realft
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b,w
REAL(SP), INTENT(OUT) :: cosint,sinint
INTERFACE
    FUNCTION func(x)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
INTEGER(I4B), PARAMETER :: M=64,NDFT=1024,MPOL=6
```

Example subroutine illustrating how to use the routine dftcor. The user supplies an external function func that returns the quantity $h(t)$. The routine then returns $\int_a^b \cos(\omega t)h(t)\,dt$ as cosint and $\int_a^b \sin(\omega t)h(t)\,dt$ as sinint.

Parameters: The values of M, NDFT, and MPOL are merely illustrative and should be optimized for your particular application. M is the number of subintervals, NDFT is the length of the FFT (a power of 2), and MPOL is the degree of polynomial interpolation used to obtain the desired frequency from the FFT.

```
INTEGER(I4B) :: nn
INTEGER(I4B), SAVE :: init=0
INTEGER(I4B), DIMENSION(MPOL) :: nnmpol
REAL(SP) :: c,cdft,cerr,corfac,corim,corre,en,s,sdft,serr
REAL(SP), SAVE :: delta
REAL(SP), DIMENSION(MPOL) :: cpol,spol,xpol
REAL(SP), DIMENSION(NDFT), SAVE :: data
REAL(SP), DIMENSION(8), SAVE :: endpts
REAL(SP), SAVE :: aold=-1.0e30_sp,bold=-1.0e30_sp
if (init /= 1 .or. a /= aold .or. b /= bold) then
    init=1
    aold=a
    bold=b
    delta=(b-a)/M
    data(1:M+1)=func(a+arth(0,1,M+1)*delta)
      Load the function values into the data array.
    data(M+2:NDFT)=0.0
    endpts(1:4)=data(1:4)
    endpts(5:8)=data(M-2:M+1)
    call realft(data(1:NDFT),1)
      realft returns the unused value corresponding to \omega_{N/2} in data(2). We actually want
      this element to contain the imaginary part corresponding to \omega_0, which is zero.
    data(2)=0.0
end if
```

Do we need to initialize, or is only ω changed?

Zero pad the rest of the data array.

Load the endpoints.

Now interpolate on the DFT result for the desired frequency. If the frequency is an ω_n, i.e., the quantity en is an integer, then cdft=data(2*en-1), sdft=data(2*en), and you could omit the interpolation.

```
en=w*delta*NDFT/TWOPI+1.0_sp
nn=min(max(int(en-0.5_sp*MPOL+1.0_sp),1),NDFT/2-MPOL+1)
nnmpol=arth(nn,1,MPOL)
cpol(1:MPOL)=data(2*nnmpol(:)-1)
spol(1:MPOL)=data(2*nnmpol(:))
xpol(1:MPOL)=nnmpol(:)
call polint(xpol,cpol,en,cdft,cerr)
call polint(xpol,spol,en,sdft,serr)
call dftcor(w,delta,a,b,endpts,corre,corim,corfac)
cdft=cdft*corfac+corre
sdft=sdft*corfac+corim
c=delta*cos(w*a)
s=delta*sin(w*a)
cosint=c*cdft-s*sdft
```

Leftmost point for the interpolation.

Now get the endpoint correction and the multiplicative factor $W(\theta)$.

Finally multiply by Δ and $\exp(i\omega a)$.

```
sinint=s*cdft+c*sdft
END SUBROUTINE dftint
```

$\star \quad \star \quad \star$

```
SUBROUTINE wt1(a,isign,wtstep)
USE nrtype; USE nrutil, ONLY : assert
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
    SUBROUTINE wtstep(a,isign)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE wtstep
END INTERFACE
```

One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm, replacing a by its wavelet transform (for isign=1), or performing the inverse operation (for isign=−1). The length of a is N, which must be an integer power of 2. The subroutine wtstep, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.

```
INTEGER(I4B) :: n,nn
n=size(a)
call assert(iand(n,n-1)==0, 'n must be a power of 2 in wt1')
if (n < 4) RETURN
if (isign >= 0) then          Wavelet transform.
    nn=n                      Start at largest hierarchy,
    do
        if (nn < 4) exit
        call wtstep(a(1:nn),isign)
        nn=nn/2               and work towards smallest.
    end do
else                          Inverse wavelet transform.
    nn=4                      Start at smallest hierarchy,
    do
        if (nn > n) exit
        call wtstep(a(1:nn),isign)
        nn=nn*2               and work towards largest.
    end do
end if
END SUBROUTINE wt1
```

```
SUBROUTINE daub4(a,isign)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
```

Applies the Daubechies 4-coefficient wavelet filter to data vector a (for isign=1) or applies its transpose (for isign=−1). Used hierarchically by routines wt1 and wtn.

```
REAL(SP), DIMENSION(size(a)) :: wksp
REAL(SP), PARAMETER :: C0=0.4829629131445341_sp,&
    C1=0.8365163037378079_sp,C2=0.2241438680420134_sp,&
    C3=-0.1294095225512604_sp
INTEGER(I4B) :: n,nh,nhp,nhm
n=size(a)
if (n < 4) RETURN
nh=n/2
nhp=nh+1
nhm=nh-1
```

```
if (isign >= 0) then          Apply filter.
    wksp(1:nhm) = C0*a(1:n-3:2)+C1*a(2:n-2:2) &
        +C2*a(3:n-1:2)+C3*a(4:n:2)
    wksp(nh)=C0*a(n-1)+C1*a(n)+C2*a(1)+C3*a(2)
    wksp(nhp:n-1) = C3*a(1:n-3:2)-C2*a(2:n-2:2) &
        +C1*a(3:n-1:2)-C0*a(4:n:2)
    wksp(n)=C3*a(n-1)-C2*a(n)+C1*a(1)-C0*a(2)
else                          Apply transpose filter.
    wksp(1)=C2*a(nh)+C1*a(n)+C0*a(1)+C3*a(nhp)
    wksp(2)=C3*a(nh)-C0*a(n)+C1*a(1)-C2*a(nhp)
    wksp(3:n-1:2) = C2*a(1:nhm)+C1*a(nhp:n-1) &
        +C0*a(2:nh)+C3*a(nh+2:n)
    wksp(4:n:2) = C3*a(1:nhm)-C0*a(nhp:n-1) &
        +C1*a(2:nh)-C2*a(nh+2:n)
end if
a(1:n)=wksp(1:n)
END SUBROUTINE daub4
```

```
MODULE pwtcom
USE nrtype
INTEGER(I4B), SAVE :: ncof=0,ioff,joff        These module variables communicate the
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: cc,cr      filter to pwt.
END MODULE pwtcom
```

```
SUBROUTINE pwtset(n)
USE nrtype; USE nrutil, ONLY : nrerror
USE pwtcom
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
```

Initializing routine for pwt, here implementing the Daubechies wavelet filters with 4, 12, and 20 coefficients, as selected by the input value n. Further wavelet filters can be included in the obvious manner. This routine must be called (once) before the first use of pwt. (For the case n=4, the specific routine daub4 is considerably faster than pwt.)

```
REAL(SP) :: sig
REAL(SP), PARAMETER :: &
    c4(4)=(/&
    0.4829629131445341_sp, 0.8365163037378079_sp, &
    0.2241438680420134_sp,-0.1294095225512604_sp /), &
    c12(12)=(/&
    0.111540743350_sp, 0.494623890398_sp, 0.751133908021_sp, &
    0.315250351709_sp,-0.226264693965_sp,-0.129766867567_sp, &
    0.097501605587_sp, 0.027522865530_sp,-0.031582039318_sp, &
    0.000553842201_sp, 0.004777257511_sp,-0.001077301085_sp /), &
    c20(20)=(/&
    0.026670057901_sp, 0.188176800078_sp, 0.527201188932_sp, &
    0.688459039454_sp, 0.281172343661_sp,-0.249846424327_sp, &
    -0.195946274377_sp, 0.127369340336_sp, 0.093057364604_sp, &
    -0.071394147166_sp,-0.029457536822_sp, 0.033212674059_sp, &
    0.003606553567_sp,-0.010733175483_sp, 0.001395351747_sp, &
    0.001992405295_sp,-0.000685856695_sp,-0.000116466855_sp, &
    0.000093588670_sp,-0.000013264203_sp /)
if (allocated(cc)) deallocate(cc)
if (allocated(cr)) deallocate(cr)
allocate(cc(n),cr(n))
ncof=n
ioff=-n/2                     These values center the "support" of the wavelets at each
joff=-n/2                         level. Alternatively, the "peaks" of the wavelets can
sig=-1.0                          be approximately centered by the choices ioff=-2
select case(n)                    and joff=-n+2. Note that daub4 and pwtset with
    case(4)                       n=4 use different default centerings.
```

```
        cc=c4
    case(12)
        cc=c12
    case(20)
        cc=c20
    case default
        call nrerror('unimplemented value n in pwtset')
end select
cr(n:1:-1) = cc
cr(n:1:-2) = -cr(n:1:-2)
END SUBROUTINE pwtset
```

f90 Here we need to have as global variables arrays whose dimensions are known only at run time. At first sight the situation is the same as with the module `fminln` in `newt` on p. 1197. If you review the discussion there and in §21.5, you will recall that there are two good ways to implement this: with allocatable arrays ("Method 1") or with pointers ("Method 2"). There is a difference here that makes allocatable arrays simpler. We do not wish to deallocate the arrays on exiting `pwtset`. On the contrary, the values in `cc` and `cr` need to be preserved for use in `pwt`. Since allocatable arrays are born in the well-defined state of "not currently allocated," we can declare the arrays here as

```
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: cc,cr
```

and test whether they were used on a previous call with

```
if (allocated(cc)) deallocate(cc)
if (allocated(cr)) deallocate(cr)
```

We are then ready to allocate the new storage:

```
allocate(cc(n),cr(n))
```

With pointers, we would need the additional machinery of nullifying the pointers on the initial call, since pointers are born in an undefined state (see §21.5).

There is an additional important point in this example. The module variables need to be used by a "sibling" routine, `pwt`. We need to be sure that they do not become undefined when we exit `pwtset`. We could ensure this by putting a `USE pwtcom` in the main program that calls both `pwtset` and `pwt`, but it's easy to forget to do this. It is preferable to put explicit `SAVE`s on all the module variables.

```
SUBROUTINE pwt(a,isign)
USE nrtype; USE nrutil, ONLY : arth,nrerror
USE pwtcom
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), INTENT(IN) :: isign
```

Partial wavelet transform: applies an arbitrary wavelet filter to data vector `a` (for `isign=1`) or applies its transpose (for `isign=-1`). Used hierarchically by routines `wt1` and `wtn`. The actual filter is determined by a preceding (and required) call to `pwtset`, which initializes the module `pwtcom`.

```
REAL(SP), DIMENSION(size(a)) :: wksp
INTEGER(I4B), DIMENSION(size(a)/2) :: jf,jr
INTEGER(I4B) :: k,n,nh,nmod
n=size(a)
if (n < 4) RETURN
if (ncof == 0) call nrerror('pwt: must call pwtset before pwt')
nmod=ncof*n
```

A positive constant equal to zero mod `n`.

```
nh=n/2
wksp(:)=0.0
jf=iand(n-1,arth(2+nmod+ioff,2,nh))        Use bitwise AND to wrap-around the point-
jr=iand(n-1,arth(2+nmod+joff,2,nh))            ers. n-1 is a mask of all bits, since n is
do k=1,ncof                                    a power of 2.
    if (isign >= 0) then                   Apply filter.
        wksp(1:nh)=wksp(1:nh)+cc(k)*a(jf+1)
        wksp(nh+1:n)=wksp(nh+1:n)+cr(k)*a(jr+1)
    else                                   Apply transpose filter.
        wksp(jf+1)=wksp(jf+1)+cc(k)*a(1:nh)
        wksp(jr+1)=wksp(jr+1)+cr(k)*a(nh+1:n)
    end if
    if (k == ncof) exit
    jf=iand(n-1,jf+1)
    jr=iand(n-1,jr+1)
end do
a(:)=wksp(:)                               Copy the results back from workspace.
END SUBROUTINE pwt
```

⋆ ⋆ ⋆

```
SUBROUTINE wtn(a,nn,isign,wtstep)
USE nrtype; USE nrutil, ONLY : arth,assert
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
INTEGER(I4B), INTENT(IN) :: isign
INTERFACE
    SUBROUTINE wtstep(a,isign)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE wtstep
END INTERFACE
```

Replaces a by its N-dimensional discrete wavelet transform, if isign is input as 1. nn is an integer array of length N, containing the lengths of each dimension (number of real values), which must all be powers of 2. a is a real array of length equal to the product of these lengths, in which the data are stored as in a multidimensional real FORTRAN array. If isign is input as -1, a is replaced by its inverse wavelet transform. The subroutine wtstep, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.

```
INTEGER(I4B) :: i1,i2,i3,idim,n,ndim,nnew,nprev,nt,ntot
REAL(SP), DIMENSION(:), ALLOCATABLE :: wksp
call assert(iand(nn,nn-1)==0, 'each dimension must be a power of 2 in wtn')
allocate(wksp(maxval(nn)))
ndim=size(nn)
ntot=product(nn(:))
nprev=1
do idim=1,ndim                                 Main loop over the dimensions.
    n=nn(idim)
    nnew=n*nprev
    if (n > 4) then
        do i2=0,ntot-1,nnew
            do i1=1,nprev
                i3=i1+i2
                wksp(1:n)=a(arth(i3,nprev,n))  Copy the relevant row or column
                i3=i3+n*nprev                      or etc. into workspace.
                if (isign >= 0) then           Do one-dimensional wavelet trans-
                    nt=n                           form.
                    do
```

```
                    if (nt < 4) exit
                    call wtstep(wksp(1:nt),isign)
                    nt=nt/2
                end do
            else                                    Or inverse transform.
                nt=4
                do
                    if (nt > n) exit
                    call wtstep(wksp(1:nt),isign)
                    nt=nt*2
                end do
            end if
            i3=i1+i2
            a(arth(i3,nprev,n))=wksp(1:n)           Copy back from workspace.
            i3=i3+n*nprev
        end do
    end do
  end if
  nprev=nnew
end do
deallocate(wksp)
END SUBROUTINE wtn
```

Chapter B14. Statistical Description of Data

```
SUBROUTINE moment(data,ave,adev,sdev,var,skew,curt)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: ave,adev,sdev,var,skew,curt
REAL(SP), DIMENSION(:), INTENT(IN) :: data
```

Given an array of data, this routine returns its mean ave, average deviation adev, standard deviation sdev, variance var, skewness skew, and kurtosis curt.

```
INTEGER(I4B) :: n
REAL(SP) :: ep
REAL(SP), DIMENSION(size(data)) :: p,s
n=size(data)
if (n <= 1) call nrerror('moment: n must be at least 2')
ave=sum(data(:))/n                  First pass to get the mean.
s(:)=data(:)-ave                    Second pass to get the first (absolute), second, third, and
ep=sum(s(:))                            fourth moments of the deviation from the mean.
adev=sum(abs(s(:)))/n
p(:)=s(:)*s(:)
var=sum(p(:))
p(:)=p(:)*s(:)
skew=sum(p(:))
p(:)=p(:)*s(:)
curt=sum(p(:))
var=(var-ep**2/n)/(n-1)             Corrected two-pass formula.
sdev=sqrt(var)
if (var /= 0.0) then
    skew=skew/(n*sdev**3)
    curt=curt/(n*var**2)-3.0_sp
else
    call nrerror('moment: no skew or kurtosis when zero variance')
end if
END SUBROUTINE moment
```

⋆ ⋆ ⋆

```
SUBROUTINE ttest(data1,data2,t,prob)
USE nrtype
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
```

Given the arrays data1 and data2, which need not have the same length, this routine returns Student's t as t, and its significance as prob, small values of prob indicating that

the arrays have significantly different means. The data arrays are assumed to be drawn from populations with the same true variance.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df,var,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
df=n1+n2-2                                          Degrees of freedom.
var=((n1-1)*var1+(n2-1)*var2)/df                    Pooled variance.
t=(ave1-ave2)/sqrt(var*(1.0_sp/n1+1.0_sp/n2))
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))           See equation (6.4.9).
END SUBROUTINE ttest
```

⋆ ⋆ ⋆

```
SUBROUTINE avevar(data,ave,var)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data
REAL(SP), INTENT(OUT) :: ave,var
```

Given array data, returns its mean as ave and its variance as var.

```
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(data)) :: s
n=size(data)
ave=sum(data(:))/n
s(:)=data(:)-ave
var=dot_product(s,s)
var=(var-sum(s)**2/n)/(n-1)          Corrected two-pass formula (14.1.8).
END SUBROUTINE avevar
```

⋆ ⋆ ⋆

```
SUBROUTINE tutest(data1,data2,t,prob)
USE nrtype
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
```

Given the arrays data1 and data2, which need not have the same length, this routine returns Student's t as t, and its significance as prob, small values of prob indicating that the arrays have significantly different means. The data arrays are allowed to be drawn from populations with unequal variances.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
t=(ave1-ave2)/sqrt(var1/n1+var2/n2)
df=(var1/n1+var2/n2)**2/((var1/n1)**2/(n1-1)+(var2/n2)**2/(n2-1))
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))
END SUBROUTINE tutest
```

⋆ ⋆ ⋆

```
SUBROUTINE tptest(data1,data2,t,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
```

Given the paired arrays data1 and data2 of the same length, this routine returns Student's t for paired data as t, and its significance as prob, small values of prob indicating a significant difference of means.

```
INTEGER(I4B) :: n
REAL(SP) :: ave1,ave2,cov,df,sd,var1,var2
n=assert_eq(size(data1),size(data2),'tptest')
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
cov=dot_product(data1(:)-ave1,data2(:)-ave2)
df=n-1
cov=cov/df
sd=sqrt((var1+var2-2.0_sp*cov)/n)
t=(ave1-ave2)/sd
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))
END SUBROUTINE tptest
```

⋆ ⋆ ⋆

```
SUBROUTINE ftest(data1,data2,f,prob)
USE nrtype
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: f,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
```

Given the arrays data1 and data2, which need not have the same length, this routine returns the value of f, and its significance as prob. Small values of prob indicate that the two arrays have significantly different variances.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df1,df2,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
if (var1 > var2) then          Make F the ratio of the larger variance to the smaller one.
    f=var1/var2
    df1=n1-1
    df2=n2-1
else
    f=var2/var1
    df1=n2-1
    df2=n1-1
end if
prob=2.0_sp*betai(0.5_sp*df2,0.5_sp*df1,df2/(df2+df1*f))
if (prob > 1.0) prob=2.0_sp-prob
END SUBROUTINE ftest
```

⋆ ⋆ ⋆

```
SUBROUTINE chsone(bins,ebins,knstrn,df,chsq,prob)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df,chsq,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: bins,ebins
```

Given the same-size arrays bins containing the observed numbers of events, and ebins containing the expected numbers of events, and given the number of constraints knstrn (normally one), this routine returns (trivially) the number of degrees of freedom df, and (nontrivially) the chi-square chsq and the significance prob. A small value of prob indicates a significant difference between the distributions bins and ebins. Note that bins and ebins are both real arrays, although bins will normally contain integer values.

```
INTEGER(I4B) :: ndum
ndum=assert_eq(size(bins),size(ebins),'chsone')
if (any(ebins(:) <= 0.0)) call nrerror('bad expected number in chsone')
df=size(bins)-knstrn
chsq=sum((bins(:)-ebins(:))**2/ebins(:))
prob=gammq(0.5_sp*df,0.5_sp*chsq)          Chi-square probability function. See §6.2.
END SUBROUTINE chsone
```

```
SUBROUTINE chstwo(bins1,bins2,knstrn,df,chsq,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df,chsq,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: bins1,bins2
```

Given the same-size arrays bins1 and bins2, containing two sets of binned data, and given the number of constraints knstrn (normally 1 or 0), this routine returns the number of degrees of freedom df, the chi-square chsq, and the significance prob. A small value of prob indicates a significant difference between the distributions bins1 and bins2. Note that bins1 and bins2 are both real arrays, although they will normally contain integer values.

```
INTEGER(I4B) :: ndum
LOGICAL(LGT), DIMENSION(size(bins1)) :: nzeromask
ndum=assert_eq(size(bins1),size(bins2),'chstwo')
nzeromask = bins1(:) /= 0.0 .or. bins2(:) /= 0.0
chsq=sum((bins1(:)-bins2(:))**2/(bins1(:)+bins2(:)),mask=nzeromask)
df=count(nzeromask)-knstrn                 No data means one less degree of freedom.
prob=gammq(0.5_sp*df,0.5_sp*chsq)          Chi-square probability function. See §6.2.
END SUBROUTINE chstwo
```

f90 nzeromask=...chisq=sum(...mask=nzeromask) We use the optional argument mask in sum to select out the elements to be summed over. In this case, at least one of the elements of bins1 or bins2 is not zero for each term in the sum.

⋆ ⋆ ⋆

```
SUBROUTINE ksone(data,func,d,prob)
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : probks,sort
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: d,prob
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
```

Given an array data, and given a user-supplied function of a single variable func which is a cumulative distribution function ranging from 0 (for smallest values of its argument) to 1 (for largest values of its argument), this routine returns the K–S statistic d, and the significance level prob. Small values of prob show that the cumulative distribution function of data is significantly different from func. The array data is modified by being sorted into ascending order.

```
INTEGER(I4B) :: n
REAL(SP) :: en
REAL(SP), DIMENSION(size(data)) :: fvals
REAL(SP), DIMENSION(size(data)+1) :: temp
call sort(data)                                  If the data are already sorted into ascending order, then this call can be omitted.
n=size(data)
en=n
fvals(:)=func(data(:))
temp=arth(0,1,n+1)/en
d=maxval(max(abs(temp(1:n)-fvals(:)), &          Compute the maximum distance between the data's c.d.f. and the user-supplied function.
    abs(temp(2:n+1)-fvals(:))))
en=sqrt(en)
prob=probks((en+0.12_sp+0.11_sp/en)*d)           Compute significance.
END SUBROUTINE ksone
```

f90 `d=maxval(max...` Note the difference between `max` and `maxval`: `max` takes two or more arguments and returns the maximum. If the arguments are two arrays, it returns an array each of whose elements is the maximum of the corresponding elements in the two arrays. `maxval` takes a single array argument and returns its maximum value.

```
SUBROUTINE kstwo(data1,data2,d,prob)
USE nrtype; USE nrutil, ONLY : cumsum
USE nr, ONLY : probks,sort2
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: d,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
```

Given arrays data1 and data2, which can be of different length, this routine returns the K–S statistic d, and the significance level prob for the null hypothesis that the data sets are drawn from the same distribution. Small values of prob show that the cumulative distribution function of data1 is significantly different from that of data2. The arrays data1 and data2 are not modified.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: en1,en2,en
REAL(SP), DIMENSION(size(data1)+size(data2)) :: dat,org
n1=size(data1)
n2=size(data2)
en1=n1
en2=n2
dat(1:n1)=data1                                  Copy the two data sets into a single array.
dat(n1+1:)=data2
```

```
org(1:n1)=0.0                                   Define an array that contains 0 when the
org(n1+1:)=1.0                                      corresponding element comes from
call sort2(dat,org)                                 data1, 1 from data2.
  Sort the array of 1's and 0's into the order of the merged data sets.
d=maxval(abs(cumsum(org)/en2-cumsum(1.0_sp-org)/en1))
  Now use cumsum to get the c.d.f. corresponding to each set of data.
en=sqrt(en1*en2/(en1+en2))
prob=probks((en+0.12_sp+0.11_sp/en)*d)          Compute significance.
END SUBROUTINE kstwo
```

The problem here is how to compute the cumulative distribution function (c.d.f.) corresponding to each set of data, and then find the corresponding KS statistic, without a serial loop over the data. The trick is to define an array that contains 0 when the corresponding element comes from the first data set and 1 when it's from the second data set. Sort the array of 1's and 0's into the same order as the merged data sets. Now tabulate the partial sums of the array. Every time you encounter a 1, the partial sum increases by 1. So if you normalize the partial sums by dividing by the number of elements in the second data set, you have the c.d.f. of the second data set.

If you subtract the array of 1's and 0's from an array of all 1's, you get an array where 1 corresponds to an element in the first data set, 0 the second data set. So tabulating its partial sums and normalizing gives the c.d.f. of the first data set. As we've seen before, tabulating partial sums can be done with a parallel algorithm (`cumsum` in `nrutil`). The KS statistic is just the maximum absolute difference of the c.d.f.'s, computed in parallel with Fortran 90's `maxval` function.

```
FUNCTION probks(alam)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alam
REAL(SP) :: probks
REAL(SP), PARAMETER :: EPS1=0.001_sp,EPS2=1.0e-8_sp
INTEGER(I4B), PARAMETER :: NITER=100
   Kolmogorov-Smirnov probability function.
INTEGER(I4B) :: j
REAL(SP) :: a2,fac,term,termbf
a2=-2.0_sp*alam**2
fac=2.0
probks=0.0
termbf=0.0                      Previous term in sum.
do j=1,NITER
    term=fac*exp(a2*j**2)
    probks=probks+term
    if (abs(term) <= EPS1*termbf .or. abs(term) <= EPS2*probks) RETURN
    fac=-fac                    Alternating signs in sum.
    termbf=abs(term)
end do
probks=1.0                      Get here only by failing to converge, which implies the func-
END FUNCTION probks                 tion is very close to 1.
```

⋆ ⋆ ⋆

```
SUBROUTINE cntab1(nn,chisq,df,prob,cramrv,ccc)
USE nrtype; USE nrutil, ONLY : outerprod
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: chisq,df,prob,cramrv,ccc
REAL(SP), PARAMETER :: TINY=1.0e-30_sp
```

Given a two-dimensional contingency table in the form of a rectangular integer array `nn`, this routine returns the chi-square `chisq`, the number of degrees of freedom `df`, the significance level `prob` (small values indicating a significant association), and two measures of association, Cramer's V (`cramrv`), and the contingency coefficient C (`ccc`).

```
INTEGER(I4B) :: nni,nnj
REAL(SP) :: sumn
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj
REAL(SP), DIMENSION(size(nn,1),size(nn,2)) :: expctd
sumi(:)=sum(nn(:,:),dim=2)                Get the row totals.
sumj(:)=sum(nn(:,:),dim=1)                Get the column totals.
sumn=sum(sumi(:))                         Get the grand total.
nni=size(sumi)-count(sumi(:) == 0.0)
```

Eliminate any zero rows by reducing the number of rows.

```
nnj=size(sumj)-count(sumj(:) == 0.0)      Eliminate any zero columns.
df=nni*nnj-nni-nnj+1                      Corrected number of degrees of freedom.
expctd(:,:)=outerprod(sumi(:),sumj(:))/sumn
chisq=sum((nn(:,:)-expctd(:,:))**2/(expctd(:,:)+TINY))
```

Do the chi-square sum. Here `TINY` guarantees that any eliminated row or column will not contribute to the sum.

```
prob=gammq(0.5_sp*df,0.5_sp*chisq)        Chi-square probability function.
cramrv=sqrt(chisq/(sumn*min(nni-1,nnj-1)))
ccc=sqrt(chisq/(chisq+sumn))
END SUBROUTINE cntab1
```

f90 `sumi(:)=sum(...dim=2)...sumj(:)=sum(...dim=1)` We use the optional argument `dim` of `sum` to sum first over the columns (`dim=2`) to get the row totals, and then to sum over the rows (`dim=1`) to get the column totals.

`expctd(:,:)=...` This is a direct implementation of equation (14.4.2) using `outerprod` from `nrutil`.

`chisq=...` And here is a direct implementation of equation (14.4.3).

```
SUBROUTINE cntab2(nn,h,hx,hy,hygx,hxgy,uygx,uxgy,uxy)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: h,hx,hy,hygx,hxgy,uygx,uxgy,uxy
REAL(SP), PARAMETER :: TINY=1.0e-30_sp
```

Given a two-dimensional contingency table in the form of a rectangular integer array `nn`, where the first index labels the x-variable and the second index labels the y variable, this routine returns the entropy `h` of the whole table, the entropy `hx` of the x-distribution, the entropy `hy` of the y-distribution, the entropy `hygx` of y given x, the entropy `hxgy` of x given y, the dependency `uygx` of y on x (eq. 14.4.15), the dependency `uxgy` of x on y (eq. 14.4.16), and the symmetrical dependency `uxy` (eq. 14.4.17).

```
REAL(SP) :: sumn
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj
sumi(:)=sum(nn(:,:),dim=2)                Get the row totals.
sumj(:)=sum(nn(:,:),dim=1)                Get the column totals.
sumn=sum(sumi(:))
hx=-sum(sumi(:)*log(sumi(:)/sumn), mask=(sumi(:) /= 0.0) )/sumn
```

Entropy of the x distribution,

```
hy=-sum(sumj(:)*log(sumj(:)/sumn), mask=(sumj(:) /= 0.0) )/sumn
```

and of the y distribution.

```
h=-sum(nn(:,:)*log(nn(:,:)/sumn), mask=(nn(:,:) /= 0) )/sumn
```

Total entropy: loop over both x and y.

```
hygx=h-hx                                  Uses equation (14.4.18),
hxgy=h-hy                                  as does this.
uygx=(hy-hygx)/(hy+TINY)                   Equation (14.4.15).
uxgy=(hx-hxgy)/(hx+TINY)                   Equation (14.4.16).
uxy=2.0_sp*(hx+hy-h)/(hx+hy+TINY)          Equation (14.4.17).
END SUBROUTINE cntab2
```

This code exploits both the `dim` feature of `sum` (see discussion after `cntab1`) and the `mask` feature to restrict the elements to be summed over.

⋆ ⋆ ⋆

```
SUBROUTINE pearsn(x,y,r,prob,z)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : betai
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: r,prob,z
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), PARAMETER :: TINY=1.0e-20_sp
```

Given two arrays `x` and `y` of the same size, this routine computes their correlation coefficient r (returned as `r`), the significance level at which the null hypothesis of zero correlation is disproved (`prob` whose small value indicates a significant correlation), and Fisher's z (returned as `z`), whose value can be used in further statistical tests as described above the routine in Volume 1.

Parameter: `TINY` will regularize the unusual case of complete correlation.

```
REAL(SP), DIMENSION(size(x)) :: xt,yt
REAL(SP) :: ax,ay,df,sxx,sxy,syy,t
INTEGER(I4B) :: n
n=assert_eq(size(x),size(y),'pearsn')
ax=sum(x)/n                                Find the means.
ay=sum(y)/n
xt(:)=x(:)-ax                              Compute the correlation co-
yt(:)=y(:)-ay                                  efficient.
sxx=dot_product(xt,xt)
syy=dot_product(yt,yt)
sxy=dot_product(xt,yt)
r=sxy/(sqrt(sxx*syy)+TINY)
z=0.5_sp*log(((1.0_sp+r)+TINY)/((1.0_sp-r)+TINY))   Fisher's z transformation.
df=n-2
t=r*sqrt(df/(((1.0_sp-r)+TINY)*((1.0_sp+r)+TINY)))  Equation (14.5.5).
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))           Student's t probability.
! prob=erfcc(abs(z*sqrt(n-1.0_sp))/SQRT2)
```

For large `n`, this easier computation of `prob`, using the short routine `erfcc`, would give approximately the same value.

```
END SUBROUTINE pearsn
```

⋆ ⋆ ⋆

```
SUBROUTINE spear(data1,data2,d,zd,probd,rs,probrs)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : betai,erfcc,sort2
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: d,zd,probd,rs,probrs
```

Given two data arrays of the same size, `data1` and `data2`, this routine returns their sum-squared difference of ranks as D, the number of standard deviations by which D deviates from its null-hypothesis expected value as `zd`, the two-sided significance level of this deviation as `probd`, Spearman's rank correlation r_s as `rs`, and the two-sided significance level of its deviation from zero as `probrs`. `data1` and `data2` are not modified. A small value of either `probd` or `probrs` indicates a significant correlation (`rs` positive) or anticorrelation (`rs` negative).

```
INTEGER(I4B) :: n
REAL(SP) :: aved,df,en,en3n,fac,sf,sg,t,vard
REAL(SP), DIMENSION(size(data1)) :: wksp1,wksp2
n=assert_eq(size(data1),size(data2),'spear')
wksp1(:)=data1(:)
wksp2(:)=data2(:)
call sort2(wksp1,wksp2)
call crank(wksp1,sf)
call sort2(wksp2,wksp1)
call crank(wksp2,sg)
wksp1(:)=wksp1(:)-wksp2(:)
d=dot_product(wksp1,wksp1)
```

Sort each of the data arrays, and convert the entries to ranks. The values `sf` and `sg` return the sums $\sum(f_k^3-f_k)$ and $\sum(g_m^3-g_m)$, respectively.

Sum the squared difference of ranks.

```
en=n
en3n=en**3-en
aved=en3n/6.0_sp-(sf+sg)/12.0_sp
fac=(1.0_sp-sf/en3n)*(1.0_sp-sg/en3n)
vard=((en-1.0_sp)*en**2*(en+1.0_sp)**2/36.0_sp)*fac
zd=(d-aved)/sqrt(vard)
probd=erfcc(abs(zd)/SQRT2)
rs=(1.0_sp-(6.0_sp/en3n)*(d+(sf+sg)/12.0_sp))/sqrt(fac)
fac=(1.0_sp+rs)*(1.0_sp-rs)
if (fac > 0.0) then
    t=rs*sqrt((en-2.0_sp)/fac)
    df=en-2.0_sp
    probrs=betai(0.5_sp*df,0.5_sp,df/(df+t**2))
else
    probrs=0.0
end if
```

Expectation value of D, and variance of D give number of standard deviations, and significance.

Rank correlation coefficient, and its t value, give its significance.

```
CONTAINS
SUBROUTINE crank(w,s)
USE nrtype; USE nrutil, ONLY : arth,array_copy
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: s
REAL(SP), DIMENSION(:), INTENT(INOUT) :: w
```

Given a sorted array `w`, replaces the elements by their rank, including midranking of ties, and returns as `s` the sum of f^3-f, where f is the number of elements in each tie.

```
INTEGER(I4B) :: i,n,ndum,nties
INTEGER(I4B), DIMENSION(size(w)) :: tstart,tend,tie,idx
n=size(w)
idx(:)=arth(1,1,n)
tie(:)=merge(1,0,w == eoshift(w,-1))
```

Index vector.

Look for ties: Compare each element to the one before. If it's equal, it's part of a tie, and we put 1 into `tie`. Otherwise we put 0.

```
tie(1)=0
w(:)=idx(:)
if (all(tie == 0)) then
    s=0.0
    RETURN
end if
call array_copy(pack(idx(:),tie(:)<eoshift(tie(:),1)),tstart,nties,ndum)
```

Boundary; the first element must be zero.

Assign ranks ignoring possible ties.

No ties—we're done.

```
  Look for 0 → 1 transitions in tie, which mean that the 0 element is the start of a tie run.
  Store index of each transition in tstart. nties is the number of ties found.
tend(1:nties)=pack(idx(:),tie(:)>eoshift(tie(:),1))
  Look for 1 → 0 transitions in tie, which mean that the 1 element is the end of a tie run.
do i=1,nties                                          Midrank assignments.
   w(tstart(i):tend(i))=(tstart(i)+tend(i))/2.0_sp
end do
tend(1:nties)=tend(1:nties)-tstart(1:nties)+1         Now calculate s.
s=sum(tend(1:nties)**3-tend(1:nties))
END SUBROUTINE crank
END SUBROUTINE spear
```

To understand how the parallel version of `crank` works, let's consider an example of 9 elements in the array `w`, which is input in sorted order to `crank`. The elements in our example are given in the second line of the following table:

index	1	2	3	4	5	6	7	8	9	
data in `w`	0	0	1	1	1	2	3	4	4	
shift right	0	0	0	1	1	1	2	3	4	
compare	1	1	0	1	1	0	0	0	1	
tie array	0	1	0	1	1	0	0	0	1	
shift left	1	0	1	1	0	0	0	1	0	
$0 \rightarrow 1$	1		3					8		start index
$1 \rightarrow 0$		2			5				9	stop index

We look for ties by comparing this array with itself, right shifted by one element ("shift right" in table). We record a 1 for each element that is the same, a 0 for each element that is different ("compare"). A 1 indicates the element is part of a tie with the *preceding* element, so we always set the first element to 0, even if it was a 1 as in our example. This gives the "tie array." Now wherever the tie array makes a transition $0 \rightarrow 1$ indicates the start of a tie run, while a $1 \rightarrow 0$ transition indicates the end of a tie run. We find these transitions by comparing the tie array to itself left shifted by one ("shift left"). If the tie array element is smaller than the shifted array element, we have a $0 \rightarrow 1$ transition and we record the corresponding index as the start of a tie. Similarly if the tie array element is larger we record the index as the end of a tie. Note that the shifts must be end-off shifts with zeros inserted in the gaps for the boundary conditions to work.

```
call array_copy(pack(idx(:),tie(:)<eoshift(tie(:),1)),
   tstart,nties,ndum)
```

The start indices (1, 3, and 8 in our example above) are here packed into the first few elements of `tstart`. `array_copy` is a useful routine in `nrutil` for copying elements from one array to another, when the number of elements to be copied is not known in advance. This line of code is equivalent to

```
tstart(:)=0
tstart(:)=pack(idx(:), tie(:) < eoshift(tie(:),1),tstart(:))
nties=count(tstart(:) > 0)
```

The point is that we don't know how many elements `pack` is going to select. We have to make sure the dimensions of both sides of the `pack` statement are the same,

so we set the optional third argument of pack to tstart. We then make a separate pass through tstart to count how many elements we copied. Alternatively, we could have used an additional logical array mask and coded this as

```
mask(:)=tie(:) < eoshift(tie(:),1)
nties=count(mask)
tstart(1:nties)=pack(idx(:),mask)
```

But we still need two passes through the mask array. The beauty of the array_copy routine is that nties is determined from the *size* of the first argument, without the necessity for a second pass through the array.

⋆ ⋆ ⋆

```
SUBROUTINE kendl1(data1,data2,tau,z,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : erfcc
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: tau,z,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
```

Given same-size data arrays data1 and data2, this program returns Kendall's τ as tau, its number of standard deviations from zero as z, and its two-sided significance level as prob. Small values of prob indicate a significant correlation (tau positive) or anticorrelation (tau negative).

```
INTEGER(I4B) :: is,j,n,n1,n2
REAL(SP) :: var
REAL(SP), DIMENSION(size(data1)) :: a1,a2
n=assert_eq(size(data1),size(data2),'kendl1')
n1=0                                    This will be the argument of one square root in (14.6.8),
n2=0                                    and this the other.
is=0                                    This will be the numerator in (14.6.8).
do j=1,n-1                                              For each first member of pair,
    a1(j+1:n)=data1(j)-data1(j+1:n)                     loop over second member.
    a2(j+1:n)=data2(j)-data2(j+1:n)
    n1=n1+count(a1(j+1:n) /= 0.0)
    n2=n2+count(a2(j+1:n) /= 0.0)
      Now accumulate the numerator in (14.6.8):
    is=is+count((a1(j+1:n) > 0.0 .and. a2(j+1:n) > 0.0) &
        .or. (a1(j+1:n) < 0.0 .and. a2(j+1:n) < 0.0)) - &
        count((a1(j+1:n) > 0.0 .and. a2(j+1:n) < 0.0) &
        .or. (a1(j+1:n) < 0.0 .and. a2(j+1:n) > 0.0))
end do
tau=real(is,sp)/sqrt(real(n1,sp)*real(n2,sp))           Equation (14.6.8).
var=(4.0_sp*n+10.0_sp)/(9.0_sp*n*(n-1.0_sp))            Equation (14.6.9).
z=tau/sqrt(var)
prob=erfcc(abs(z)/SQRT2)                                Significance.
END SUBROUTINE kendl1
```

```
SUBROUTINE kendl2(tab,tau,z,prob)
USE nrtype; USE nrutil, ONLY : cumsum
USE nr, ONLY : erfcc
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: tab
REAL(SP), INTENT(OUT) :: tau,z,prob
```

Given a two-dimensional table tab such that tab(k,l) contains the number of events falling in bin k of one variable and bin l of another, this program returns Kendall's τ as tau, its number of standard deviations from zero as z, and its two-sided significance level as prob. Small values of prob indicate a significant correlation (tau positive) or anticorrelation (tau

negative) between the two variables. Although tab is a real array, it will normally contain integral values.

```
REAL(SP), DIMENSION(size(tab,1),size(tab,2)) :: cum,cumt
INTEGER(I4B) :: i,j,ii,jj
REAL(SP) :: sc,sd,en1,en2,points,var
ii=size(tab,1)
jj=size(tab,2)
do i=1,ii                                          Get cumulative sums leftward along
    cumt(i,jj:1:-1)=cumsum(tab(i,jj:1:-1))             rows.
end do
en2=sum(tab(1:ii,1:jj-1)*cumt(1:ii,2:jj))          Tally the extra-y pairs.
do j=1,jj                                          Get counts of points to lower-right
    cum(ii:1:-1,j)=cumsum(cumt(ii:1:-1,j))             of each cell in cum.
end do
points=cum(1,1)                                    Total number of entries in table.
sc=sum(tab(1:ii-1,1:jj-1)*cum(2:ii,2:jj))          Tally the concordant pairs.
do j=1,jj                                          Now get counts of points to upper-
    cum(1:ii,j)=cumsum(cumt(1:ii,j))                   right of each cell in cum,
end do
sd=sum(tab(2:ii,1:jj-1)*cum(1:ii-1,2:jj))          giving tally of discordant points.
do j=1,jj                                          Finally, get cumulative sums upward
    cumt(ii:1:-1,j)=cumsum(tab(ii:1:-1,j))             along columns,
end do
en1=sum(tab(1:ii-1,1:jj)*cumt(2:ii,1:jj))          giving the count of extra-x pairs,
tau=(sc-sd)/sqrt((en1+sc+sd)*(en2+sc+sd))          and compute desired results.
var=(4.0_sp*points+10.0_sp)/(9.0_sp*points*(points-1.0_sp))
z=tau/sqrt(var)
prob=erfcc(abs(z)/SQRT2)
END SUBROUTINE kendl2
```

The underlying algorithm in kendl2 might seem to require looping over all *pairs* of cells in the two-dimensional table tab. Actually, however, clever use of the cumsum utility function reduces this to a simple loop over all the cells; moreover this "loop" parallelizes into a simple parallel product and call to the sum intrinsic. The basic idea is shown in the following table:

		d	d
	t	y	y
	x	c	c
	x	c	c
	x	c	c

Relative to the cell marked t (which we use to denote the numerical value it contains), the cells marked d contribute to the "discordant" tally in Volume 1's equation (14.6.8),

while the cells marked c contribute to the "concordant" tally. Likewise, the cells marked x and y contribute, respectively, to the "extra-x" and "extra-y" tallies. What about the cells left blank? Since we want to count pairs of cells only *once*, without duplication, these cells will be counted, relative to the location shown as t, when t itself moves into the blank-cell area.

Symbolically we have

$$\begin{aligned}
\text{concordant} &= \sum_n t_n \left(\sum_{\text{lower right}} c_m \right) \\
\text{discordant} &= \sum_n t_n \left(\sum_{\text{upper right}} d_m \right) \\
\text{extra-}x &= \sum_n t_n \left(\sum_{\text{below}} x_m \right) \\
\text{extra-}y &= \sum_n t_n \left(\sum_{\text{to the right}} y_m \right)
\end{aligned} \tag{B14.1}$$

Here n varies over all the positions in the table, while the limits of the inner sums are relative to the position of n. (The letters t_n, c_m, d_m, x_m, y_m all represent the value in a cell; we use different letters only to make the relation with the above table clear.) Now the final trick is to recognize that the inner sums, over cells to the lower- or upper-right, below, and to the right can be done in parallel by cumulative sums (`cumsum`) sweeping to the right and up. The routine does these in a nonintuitive order merely to be able to reuse maximally the scratch spaces `cum` and `cumt`.

⋆ ⋆ ⋆

```
SUBROUTINE ks2d1s(x1,y1,quadvl,d1,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : pearsn,probks,quadct
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1
REAL(SP), INTENT(OUT) :: d1,prob
INTERFACE
    SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    END SUBROUTINE quadvl
END INTERFACE
```

Two-dimensional Kolmogorov-Smirnov test of one sample against a model. Given the x- and y-coordinates of a set of data points in arrays `x1` and `y1` of the same length, and given a user-supplied function `quadvl` that exemplifies the model, this routine returns the two-dimensional K-S statistic as `d1`, and its significance level as `prob`. Small values of `prob` show that the sample is significantly different from the model. Note that the test is slightly distribution-dependent, so `prob` is only an estimate.

```
INTEGER(I4B) :: j,n1
REAL(SP) :: dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,rr,sqen
n1=assert_eq(size(x1),size(y1),'ks2d1s')
d1=0.0
```

```
do j=1,n1                                   Loop over the data points.
    call quadct(x1(j),y1(j),x1,y1,fa,fb,fc,fd)
    call quadvl(x1(j),y1(j),ga,gb,gc,gd)
    d1=max(d1,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
      For both the sample and the model, the distribution is integrated in each of four quad-
      rants, and the maximum difference is saved.
end do
call pearsn(x1,y1,r1,dum,dumm)              Get the linear correlation coefficient r1.
sqen=sqrt(real(n1,sp))
rr=sqrt(1.0_sp-r1**2)
  Estimate the probability using the K-S probability function probks.
prob=probks(d1*sqen/(1.0_sp+rr*(0.25_sp-0.75_sp/sqen)))
END SUBROUTINE ks2d1s



SUBROUTINE quadct(x,y,xx,yy,fa,fb,fc,fd)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:), INTENT(IN) :: xx,yy
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    Given an origin (x, y), and an array of points with coordinates xx and yy, count how many of
    them are in each quadrant around the origin, and return the normalized fractions. Quadrants
    are labeled alphabetically, counterclockwise from the upper right. Used by ks2d1s and
    ks2d2s.
INTEGER(I4B) :: na,nb,nc,nd,nn
REAL(SP) :: ff
nn=assert_eq(size(xx),size(yy),'quadct')
na=count(yy(:) > y .and. xx(:) > x)
nb=count(yy(:) > y .and. xx(:) <= x)
nc=count(yy(:) <= y .and. xx(:) <= x)
nd=nn-na-nb-nc
ff=1.0_sp/nn
fa=ff*na
fb=ff*nb
fc=ff*nc
fd=ff*nd
END SUBROUTINE quadct



SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    This is a sample of a user-supplied routine to be used with ks2d1s. In this case, the model
    distribution is uniform inside the square −1 < x < 1, −1 < y < 1. In general this routine
    should return, for any point (x, y), the fraction of the total distribution in each of the
    four quadrants around that point. The fractions, fa, fb, fc, and fd, must add up to 1.
    Quadrants are alphabetical, counterclockwise from the upper right.
REAL(SP) :: qa,qb,qc,qd
qa=min(2.0_sp,max(0.0_sp,1.0_sp-x))
qb=min(2.0_sp,max(0.0_sp,1.0_sp-y))
qc=min(2.0_sp,max(0.0_sp,x+1.0_sp))
qd=min(2.0_sp,max(0.0_sp,y+1.0_sp))
fa=0.25_sp*qa*qb
fb=0.25_sp*qb*qc
fc=0.25_sp*qc*qd
fd=0.25_sp*qd*qa
END SUBROUTINE quadvl
```

```
SUBROUTINE ks2d2s(x1,y1,x2,y2,d,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : pearsn,probks,quadct
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1,x2,y2
REAL(SP), INTENT(OUT) :: d,prob
```

Compute two-dimensional Kolmogorov-Smirnov test on two samples. Input are the x- and y-coordinates of the first sample in arrays x1 and y1 of the same length, and of the second sample in arrays x2 and y2 of the same length (possibly different from the length of the first sample). The routine returns the two-dimensional, two-sample K-S statistic as d, and its significance level as prob. Small values of prob show that the two samples are significantly different. Note that the test is slightly distribution-dependent, so prob is only an estimate.

```
INTEGER(I4B) :: j,n1,n2
REAL(SP) :: d1,d2,dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,r2,rr,sqen
n1=assert_eq(size(x1),size(y1),'ks2d2s: n1')
n2=assert_eq(size(x2),size(y2),'ks2d2s: n2')
d1=0.0
do j=1,n1                                   First, use points in the first sample as origins.
    call quadct(x1(j),y1(j),x1,y1,fa,fb,fc,fd)
    call quadct(x1(j),y1(j),x2,y2,ga,gb,gc,gd)
    d1=max(d1,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
end do
d2=0.0
do j=1,n2                                   Then, use points in the second sample as ori-
    call quadct(x2(j),y2(j),x1,y1,fa,fb,fc,fd)      gins.
    call quadct(x2(j),y2(j),x2,y2,ga,gb,gc,gd)
    d2=max(d2,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
end do
d=0.5_sp*(d1+d2)                            Average the K-S statistics.
sqen=sqrt(real(n1,sp)*real(n2,sp)/real(n1+n2,sp))
call pearsn(x1,y1,r1,dum,dumm)              Get the linear correlation coefficient for each sam-
call pearsn(x2,y2,r2,dum,dumm)                  ple.
rr=sqrt(1.0_sp-0.5_sp*(r1**2+r2**2))
```

Estimate the probability using the K-S probability function probks.

```
prob=probks(d*sqen/(1.0_sp+rr*(0.25_sp-0.75_sp/sqen)))
END SUBROUTINE ks2d2s
```

⋆ ⋆ ⋆

```
FUNCTION savgol(nl,nrr,ld,m)
USE nrtype; USE nrutil, ONLY : arth,assert,poly
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
```

Returns in array c, in wrap-around order (N.B.!) consistent with the argument respns in routine convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data points used, while nrr is the number of rightward (future) data points, making the total number of data points used $\mathtt{nl}+\mathtt{nrr}+1$. ld is the order of the derivative desired (e.g., $\mathtt{ld}=0$ for smoothed function). m is the order of the smoothing polynomial, also equal to the highest conserved moment; usual value is $\mathtt{m}=2$ or $\mathtt{m}=4$.

```
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B) :: imj,ipj,mm,np
INTEGER(I4B), DIMENSION(m+1) :: indx
REAL(SP) :: d,sm
REAL(SP), DIMENSION(m+1) :: b
REAL(SP), DIMENSION(m+1,m+1) :: a
INTEGER(I4B) :: irng(nl+nrr+1)
call assert(nl >= 0, nrr >= 0, ld <= m, nl+nrr >= m, 'savgol args')
do ipj=0,2*m                                Set up the normal equations of the desired least
    sm=sum(arth(1.0_sp,1.0_sp,nrr)**ipj)+&      squares fit.
        sum(arth(-1.0_sp,-1.0_sp,nl)**ipj)
```

```
    if (ipj == 0) sm=sm+1.0_sp
    mm=min(ipj,2*m-ipj)
    do imj=-mm,mm,2
        a(1+(ipj+imj)/2,1+(ipj-imj)/2)=sm
    end do
end do
call ludcmp(a(:,:),indx(:),d)          Solve them: LU decomposition.
b(:)=0.0
b(ld+1)=1.0                            Right-hand-side vector is unit vector, depending
call lubksb(a(:,:),indx(:),b(:))           on which derivative we want.
```

Backsubstitute, giving one row of the inverse matrix.

```
savgol(:)=0.0                          Zero the output array (it may be bigger than
irng(:)=arth(-nl,1,nrr+nl+1)               number of coefficients).
np=nl+nrr+1
savgol(mod(np-irng(:),np)+1)=poly(real(irng(:),sp),b(:))
```

Each Savitzky-Golay coefficient is the value of the polynomial in (14.8.6) at the corresponding integer. The polynomial coefficients are a row of the inverse matrix. The `mod` function takes care of the wrap-around order.

```
END FUNCTION savgol
```

`do imj=-mm,mm,2` Here is an example of a loop that cannot be parallelized in the framework of Fortran 90: We need to access "skew" sections of the matrix `a`.

`savgol...=poly(real(irng(:),sp),b(:)))` The `poly` function in `nrutil` returns the value of a polynomial, here the one in equation (14.8.6). We need the explicit kind type parameter `sp` in the `real` function, otherwise it would return type default real for the integer argument and would not automatically convert to double precision if desired.

Chapter B15. Modeling of Data

```
SUBROUTINE fit(x,y,a,b,siga,sigb,chi2,q,sig)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : gammq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
REAL(SP), DIMENSION(:), OPTIONAL, INTENT(IN) :: sig
```

Given a set of data points in same-size arrays `x` and `y`, fit them to a straight line $y = a + bx$ by minimizing χ^2. `sig` is an optional array of the same length containing the individual standard deviations. If it is present, then `a`,`b` are returned with their respective probable uncertainties `siga` and `sigb`, the chi-square `chi2`, and the goodness-of-fit probability `q` (that the fit would have χ^2 this large or larger). If `sig` is not present, then `q` is returned as `1.0` and the normalization of `chi2` is to unit standard deviation on all points.

```
INTEGER(I4B) :: ndata
REAL(SP) :: sigdat,ss,sx,sxoss,sy,st2
REAL(SP), DIMENSION(size(x)), TARGET :: t
REAL(SP), DIMENSION(:), POINTER :: wt
if (present(sig)) then
    ndata=assert_eq(size(x),size(y),size(sig),'fit')
    wt=>t                                          Use temporary variable t to store weights.
    wt(:)=1.0_sp/(sig(:)**2)
    ss=sum(wt(:))                                  Accumulate sums with weights.
    sx=dot_product(wt,x)
    sy=dot_product(wt,y)
else
    ndata=assert_eq(size(x),size(y),'fit')
    ss=real(size(x),sp)                            Accumulate sums without weights.
    sx=sum(x)
    sy=sum(y)
end if
sxoss=sx/ss
t(:)=x(:)-sxoss
if (present(sig)) then
    t(:)=t(:)/sig(:)
    b=dot_product(t/sig,y)
else
    b=dot_product(t,y)
end if
st2=dot_product(t,t)
b=b/st2                                            Solve for a, b, σ_a, and σ_b.
a=(sy-sx*b)/ss
siga=sqrt((1.0_sp+sx*sx/(ss*st2))/ss)
sigb=sqrt(1.0_sp/st2)
t(:)=y(:)-a-b*x(:)
q=1.0
if (present(sig)) then
    t(:)=t(:)/sig(:)
    chi2=dot_product(t,t)                          Calculate χ².
    if (ndata > 2) q=gammq(0.5_sp*(size(x)-2),0.5_sp*chi2)   Equation (15.2.12).
else
    chi2=dot_product(t,t)
```

```
    sigdat=sqrt(chi2/(size(x)-2))
    siga=siga*sigdat
    sigb=sigb*sigdat
end if
END SUBROUTINE fit
```

For unweighted data evaluate typical sig using chi2, and adjust the standard deviations.

f90 `REAL(SP), DIMENSION(:), POINTER :: wt...wt=>t` When standard deviations are supplied in `sig`, we need to compute the weights for the least squares fit in a temporary array `wt`. Later in the routine, we need another temporary array, which we call `t` to correspond to the variable in equation (15.2.15). It would be confusing to use the same name for both arrays. In Fortran 77 the arrays could share storage with an `EQUIVALENCE` declaration, but that is a deprecated feature in Fortran 90. We accomplish the same thing by making `wt` a pointer alias to `t`.

⋆ ⋆ ⋆

```
SUBROUTINE fitexy(x,y,sigx,sigy,a,b,siga,sigb,chi2,q)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
USE nr, ONLY : avevar,brent,fit,gammq,mnbrak,zbrent
USE chixyfit
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sigx,sigy
REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
REAL(SP), PARAMETER :: POTN=1.571000_sp,BIG=1.0e30_sp,ACC=1.0e-3_sp
```

Straight-line fit to input data `x` and `y` with errors in both x and y, the respective standard deviations being the input quantities `sigx` and `sigy`. `x`, `y`, `sigx`, and `sigy` are all arrays of the same length. Output quantities are `a` and `b` such that $y = a + bx$ minimizes χ^2, whose value is returned as `chi2`. The χ^2 probability is returned as `q`, a small value indicating a poor fit (sometimes indicating underestimated errors). Standard errors on `a` and `b` are returned as `siga` and `sigb`. These are not meaningful if either (i) the fit is poor, or (ii) b is so large that the data are consistent with a vertical (infinite b) line. If `siga` and `sigb` are returned as `BIG`, then the data are consistent with *all* values of b.

```
INTEGER(I4B) :: j,n
REAL(SP), DIMENSION(size(x)), TARGET :: xx,yy,sx,sy,ww
REAL(SP), DIMENSION(6) :: ang,ch
REAL(SP) :: amx,amn,varx,vary,scale,bmn,bmx,d1,d2,r2,&
    dum1,dum2,dum3,dum4,dum5
n=assert_eq(size(x),size(y),size(sigx),size(sigy),'fitexy')
xxp=>xx
yyp=>yy
sxp=>sx
syp=>sy
wwp=>ww
call avevar(x,dum1,varx)
call avevar(y,dum1,vary)
scale=sqrt(varx/vary)
xx(:)=x(:)
yy(:)=y(:)*scale
sx(:)=sigx(:)
sy(:)=sigy(:)*scale
ww(:)=sqrt(sx(:)**2+sy(:)**2)
call fit(xx,yy,dum1,b,dum2,dum3,dum4,dum5,ww)
offs=0.0
ang(1)=0.0
ang(2)=atan(b)
ang(4)=0.0
ang(5)=ang(2)
ang(6)=POTN
do j=4,6
    ch(j)=chixy(ang(j))
```

Set up communication with function `chixy` through global variables in the module `chixyfit`.

Find the x and y variances, and scale the data.

Use both x and y weights in first trial fit.

Trial fit for `b`.

Construct several angles for reference points.

Make `b` an angle.

```
end do
call mnbrak(ang(1),ang(2),ang(3),ch(1),ch(2),ch(3),chixy)
```
Bracket the χ^2 minimum and then locate it with brent.
```
chi2=brent(ang(1),ang(2),ang(3),chixy,ACC,b)
chi2=chixy(b)
a=aa
q=gammq(0.5_sp*(n-2),0.5_sp*chi2)          Compute χ² probability.
r2=1.0_sp/sum(ww(:))                       Save inverse sum of weights at the minimum.
bmx=BIG                                    Now, find standard errors for b as points where
bmn=BIG                                        Δχ² = 1.
offs=chi2+1.0_sp
do j=1,6                                   Go through saved values to bracket the de-
    if (ch(j) > offs) then                     sired roots. Note periodicity in slope an-
        d1=mod(abs(ang(j)-b),PI)               gles.
        d2=PI-d1
        if (ang(j) < b) call swap(d1,d2)
        if (d1 < bmx) bmx=d1
        if (d2 < bmn) bmn=d2
    end if
end do
if (bmx < BIG) then                        Call zbrent to find the roots.
    bmx=zbrent(chixy,b,b+bmx,ACC)-b
    amx=aa-a
    bmn=zbrent(chixy,b,b-bmn,ACC)-b
    amn=aa-a
    sigb=sqrt(0.5_sp*(bmx**2+bmn**2))/(scale*cos(b)**2)
    siga=sqrt(0.5_sp*(amx**2+amn**2)+r2)/scale      Error in a has additional piece
else                                                   r2.
    sigb=BIG
    siga=BIG
end if
a=a/scale                                  Unscale the answers.
b=tan(b)/scale
END SUBROUTINE fitexy
```

f90 `USE chixyfit` We need to pass arrays and other variables to `chixy`, but not as arguments. See §21.5 and the discussion of `fminln` on p. 1197 for two good ways to do this. The pointer construction here is analogous to the one used in `fminln`.

```
MODULE chixyfit
USE nrtype; USE nrutil, ONLY : nrerror
REAL(SP), DIMENSION(:), POINTER :: xxp,yyp,sxp,syp,wwp
REAL(SP) :: aa,offs
CONTAINS

FUNCTION chixy(bang)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: bang
REAL(SP) :: chixy
REAL(SP), PARAMETER :: BIG=1.0e30_sp
```
Captive function of `fitexy`, returns the value of $(\chi^2 - \texttt{offs})$ for the slope `b=tan(bang)`. Scaled data and `offs` are communicated via the module `chixyfit`.
```
REAL(SP) :: avex,avey,sumw,b
if (.not. associated(wwp)) call nrerror("chixy: bad pointers")
b=tan(bang)
wwp(:)=(b*sxp(:))**2+syp(:)**2
where (wwp(:) < 1.0/BIG)
    wwp(:)=BIG
elsewhere
    wwp(:)=1.0_sp/wwp(:)
end where
```

```
sumw=sum(wwp)
avex=dot_product(wwp,xxp)/sumw
avey=dot_product(wwp,yyp)/sumw
aa=avey-b*avex
chixy=sum(wwp(:)*(yyp(:)-aa-b*xxp(:))**2)-offs
END FUNCTION chixy
END MODULE chixyfit
```

⋆ ⋆ ⋆

```
SUBROUTINE lfit(x,y,sig,a,maska,covar,chisq,funcs)
USE nrtype; USE nrutil, ONLY : assert_eq,diagmult,nrerror
USE nr, ONLY :covsrt,gaussj
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: covar
REAL(SP), INTENT(OUT) :: chisq
INTERFACE
    SUBROUTINE funcs(x,arr)
    USE nrtype
    IMPLICIT NONE
    REAL(SP),INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: arr
    END SUBROUTINE funcs
END INTERFACE
```

Given a set of N data points x, y with individual standard deviations sig, all arrays of length N, use χ^2 minimization to fit for some or all of the M coefficients a of a function that depends linearly on a, $y = \sum_{i=1}^{M} \mathtt{a}_i \times \mathtt{afunc}_i(x)$. The input logical array maska of length M indicates by true entries those components of a that should be fitted for, and by false entries those components that should be held fixed at their input values. The program returns values for a, $\chi^2 = \mathtt{chisq}$, and the $M \times M$ covariance matrix covar. (Parameters held fixed will return zero covariances.) The user supplies a subroutine funcs(x,afunc) that returns the M basis functions evaluated at $x = \mathtt{x}$ in the array afunc.

```
INTEGER(I4B) :: i,j,k,l,ma,mfit,n
REAL(SP) :: sig2i,wt,ym
REAL(SP), DIMENSION(size(maska)) :: afunc
REAL(SP), DIMENSION(size(maska),1) :: beta
n=assert_eq(size(x),size(y),size(sig),'lfit: n')
ma=assert_eq(size(maska),size(a),size(covar,1),size(covar,2),'lfit: ma')
mfit=count(maska)                          Number of parameters to fit for.
if (mfit == 0) call nrerror('lfit: no parameters to be fitted')
covar(1:mfit,1:mfit)=0.0                   Initialize the (symmetric) matrix.
beta(1:mfit,1)=0.0
do i=1,n                                   Loop over data to accumulate coefficients of
    call funcs(x(i),afunc)                     the normal equations.
    ym=y(i)
    if (mfit < ma) ym=ym-sum(a(1:ma)*afunc(1:ma), mask=.not. maska)
      Subtract off dependences on known pieces of the fitting function.
    sig2i=1.0_sp/sig(i)**2
    j=0
    do l=1,ma
        if (maska(l)) then
            j=j+1
            wt=afunc(l)*sig2i
            k=count(maska(1:l))
            covar(j,1:k)=covar(j,1:k)+wt*pack(afunc(1:l),maska(1:l))
            beta(j,1)=beta(j,1)+ym*wt
        end if
    end do
```

```
end do
call diagmult(covar(1:mfit,1:mfit),0.5_sp)
covar(1:mfit,1:mfit)= &                    Fill in above the diagonal from symmetry.
    covar(1:mfit,1:mfit)+transpose(covar(1:mfit,1:mfit))
call gaussj(covar(1:mfit,1:mfit),beta(1:mfit,1:1))      Matrix solution.
a(1:ma)=unpack(beta(1:ma,1),maska,a(1:ma))
  Partition solution to appropriate coefficients a.
chisq=0.0                                  Evaluate χ² of the fit.
do i=1,n
    call funcs(x(i),afunc)
    chisq=chisq+((y(i)-dot_product(a(1:ma),afunc(1:ma)))/sig(i))**2
end do
call covsrt(covar,maska)                   Sort covariance matrix to true order of fitting
END SUBROUTINE lfit                            coefficients.
```

f90 `if (mfit < ma) ym=ym-sum(a(1:ma)*afunc(1:ma), mask=.not. maska)` This is the first of several uses of `maska` in this routine to control which elements of an array are to be used. Here we include in the sum only elements for which `maska` is false, i.e., elements corresponding to parameters that are not being fitted for.

`covar(j,1:k)=covar(j,1:k)+wt*pack(afunc(1:l),maska(1:l))` Here `maska` controls which elements of `afunc` get packed into the covariance matrix.

`call diagmult(covar(1:mfit,1:mfit),0.5_sp)` See discussion of `diagadd` after `hqr` on p. 1234.

`a(1:ma)=unpack(beta(1:ma,1),maska,a(1:ma))` And here `maska` controls which elements of `beta` get unpacked into the appropriate slots in `a`. Where `maska` is false, corresponding elements are selected from the third argument of `unpack`, here `a` itself. The net effect is that those elements remain unchanged.

⋆ ⋆ ⋆

```
SUBROUTINE covsrt(covar,maska)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: covar
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    Expand in storage the covariance matrix covar, so as to take into account parameters that
    are being held fixed. (For the latter, return zero covariances.)
INTEGER(I4B) :: ma,mfit,j,k
ma=assert_eq(size(covar,1),size(covar,2),size(maska),'covsrt')
mfit=count(maska)
covar(mfit+1:ma,1:ma)=0.0
covar(1:ma,mfit+1:ma)=0.0
k=mfit
do j=ma,1,-1
    if (maska(j)) then
        call swap(covar(1:ma,k),covar(1:ma,j))
        call swap(covar(k,1:ma),covar(j,1:ma))
        k=k-1
    end if
end do
END SUBROUTINE covsrt
```

⋆ ⋆ ⋆

```
SUBROUTINE svdfit(x,y,sig,a,v,w,chisq,funcs)
USE nrtype; USE nrutil, ONLY : assert_eq,vabs
USE nr, ONLY : svbksb,svdcmp
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:), INTENT(OUT) :: a,w
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
REAL(SP), INTENT(OUT) :: chisq
INTERFACE
    FUNCTION funcs(x,n)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: funcs
    END FUNCTION funcs
END INTERFACE
REAL(SP), PARAMETER :: TOL=1.0e-5_sp
```

Given a set of N data points `x`, `y` with individual standard deviations `sig`, all arrays of length N, use χ^2 minimization to determine the M coefficients `a` of a function that depends linearly on `a`, $y = \sum_{i=1}^{M} \mathtt{a}_i \times \mathtt{afunc}_i(x)$. Here we solve the fitting equations using singular value decomposition of the $N \times M$ matrix, as in §2.6. On output, the $M \times M$ array `v` and the vector `w` of length M define part of the singular value decomposition, and can be used to obtain the covariance matrix. The program returns values for the M fit parameters `a`, and χ^2, `chisq`. The user supplies a subroutine `funcs(x,afunc)` that returns the M basis functions evaluated at $x = X$ in the array `afunc`.

```
INTEGER(I4B) :: i,ma,n
REAL(SP), DIMENSION(size(x)) :: b,sigi
REAL(SP), DIMENSION(size(x),size(a)) :: u,usav
n=assert_eq(size(x),size(y),size(sig),'svdfit: n')
ma=assert_eq(size(a),size(v,1),size(v,2),size(w),'svdfit: ma')
sigi=1.0_sp/sig                          Accumulate coefficients of the fitting matrix in
b=y*sigi                                     u.
do i=1,n
    usav(i,:)=funcs(x(i),ma)
end do
u=usav*spread(sigi,dim=2,ncopies=ma)
usav=u
call svdcmp(u,w,v)                       Singular value decomposition.
where (w < TOL*maxval(w)) w=0.0          Edit the singular values, given TOL from the pa-
call svbksb(u,w,v,b,a)                       rameter statement.
chisq=vabs(matmul(usav,a)-b)**2          Evaluate chi-square.
END SUBROUTINE svdfit
```

f90 `u=usav*spread(sigi,dim=2,ncopies=ma)` Remember how `spread` works: the vector `sigi` is copied *along* the dimension 2, making a matrix whose columns are each a copy of `sigi`. The multiplication here is element by element, so each row of `usav` is multiplied by the corresponding element of `sigi`.

`chisq=vabs(matmul(usav,a)-b)**2` Fortran 90's `matmul` intrinsic allows us to evaluate χ^2 from the mathematical definition in terms of matrices. `vabs` in `nrutil` returns the length of a vector (L_2 norm).

```
SUBROUTINE svdvar(v,w,cvm)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: v
REAL(SP), DIMENSION(:), INTENT(IN) :: w
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: cvm
```

To evaluate the covariance matrix cvm of the fit for M parameters obtained by svdfit, call this routine with matrices v,w as returned from svdfit. The dimensions are M for w and $M \times M$ for v and cvm.

```
INTEGER(I4B) :: ma
REAL(SP), DIMENSION(size(w)) :: wti
ma=assert_eq((/size(v,1),size(v,2),size(w),size(cvm,1),size(cvm,2)/),&
    'svdvar')
where (w /= 0.0)
    wti=1.0_sp/(w*w)
elsewhere
    wti=0.0
end where
cvm=v*spread(wti,dim=1,ncopies=ma)
cvm=matmul(cvm,transpose(v))        Covariance matrix is given by (15.4.20).
END SUBROUTINE svdvar
```

`where (w /= 0.0)...elsewhere...end where` This is the standard Fortran 90 construction for doing different things to a matrix depending on some condition. Here we want to avoid inverting elements of w that are zero.

`cvm=v*spread(wti,dim=1,ncopies=ma)` Each column of v gets multiplied by the corresponding element of wti. Contrast the construction `spread(...dim=2...)` in svdfit.

⋆ ⋆ ⋆

```
FUNCTION fpoly(x,n)
USE nrtype; USE nrutil, ONLY : geop
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), DIMENSION(n) :: fpoly
```

Fitting routine for a polynomial of degree $n-1$, returning n coefficients in fpoly.

```
fpoly=geop(1.0_sp,x,n)
END FUNCTION fpoly
```

⋆ ⋆ ⋆

```
FUNCTION fleg(x,nl)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
INTEGER(I4B), INTENT(IN) :: nl
REAL(SP), DIMENSION(nl) :: fleg
```

Fitting routine for an expansion with nl Legendre polynomials evaluated at x and returned in the array fleg of length nl. The evaluation uses the recurrence relation as in §5.5.

```
INTEGER(I4B) :: j
REAL(SP) :: d,f1,f2,twox
fleg(1)=1.0
fleg(2)=x
if (nl > 2) then
    twox=2.0_sp*x
    f2=x
    d=1.0
    do j=3,nl
        f1=d
        f2=f2+twox
        d=d+1.0_sp
```

```
        fleg(j)=(f2*fleg(j-1)-f1*fleg(j-2))/d
    end do
end if
END FUNCTION fleg
```

⋆ ⋆ ⋆

```
SUBROUTINE mrqmin(x,y,sig,a,maska,covar,alpha,chisq,funcs,alamda)
USE nrtype; USE nrutil, ONLY : assert_eq,diagmult
USE nr, ONLY : covsrt,gaussj
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: covar,alpha
REAL(SP), INTENT(OUT) :: chisq
REAL(SP), INTENT(INOUT) :: alamda
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
INTERFACE
    SUBROUTINE funcs(x,a,yfit,dyda)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
    END SUBROUTINE funcs
END INTERFACE
```

Levenberg-Marquardt method, attempting to reduce the value χ^2 of a fit between a set of N data points x, y with individual standard deviations sig, and a nonlinear function dependent on M coefficients a. The input logical array maska of length M indicates by true entries those components of a that should be fitted for, and by false entries those components that should be held fixed at their input values. The program returns current best-fit values for the parameters a, and $\chi^2 =$ chisq. The $M \times M$ arrays covar and alpha are used as working space during most iterations. Supply a subroutine funcs(x,a,yfit,dyda) that evaluates the fitting function yfit, and its derivatives dyda with respect to the fitting parameters a at x. On the first call provide an initial guess for the parameters a, and set alamda<0 for initialization (which then sets alamda=.001). If a step succeeds chisq becomes smaller and alamda decreases by a factor of 10. If a step fails alamda grows by a factor of 10. You must call this routine repeatedly until convergence is achieved. Then, make one final call with alamda=0, so that covar returns the covariance matrix, and alpha the curvature matrix. (Parameters held fixed will return zero covariances.)

```
INTEGER(I4B) :: ma,ndata
INTEGER(I4B), SAVE :: mfit
call mrqmin_private
CONTAINS

SUBROUTINE mrqmin_private
REAL(SP), SAVE :: ochisq
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: atry,beta
REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: da
ndata=assert_eq(size(x),size(y),size(sig),'mrqmin: ndata')
ma=assert_eq((/size(a),size(maska),size(covar,1),size(covar,2),&
    size(alpha,1),size(alpha,2)/),'mrqmin: ma')
mfit=count(maska)
if (alamda < 0.0) then                          Initialization.
    allocate(atry(ma),beta(ma),da(ma,1))
    alamda=0.001_sp
    call mrqcof(a,alpha,beta)
    ochisq=chisq
    atry=a
end if
covar(1:mfit,1:mfit)=alpha(1:mfit,1:mfit)
call diagmult(covar(1:mfit,1:mfit),1.0_sp+alamda)
```

```
    Alter linearized fitting matrix, by augmenting diagonal elements.
da(1:mfit,1)=beta(1:mfit)
call gaussj(covar(1:mfit,1:mfit),da(1:mfit,1:1))      Matrix solution.
if (alamda == 0.0) then                               Once converged, evaluate covariance ma-
    call covsrt(covar,maska)                              trix.
    call covsrt(alpha,maska)                          Spread out alpha to its full size too.
    deallocate(atry,beta,da)
    RETURN
end if
atry=a+unpack(da(1:mfit,1),maska,0.0_sp)              Did the trial succeed?
call mrqcof(atry,covar,da(1:mfit,1))
if (chisq < ochisq) then                              Success, accept the new solution.
    alamda=0.1_sp*alamda
    ochisq=chisq
    alpha(1:mfit,1:mfit)=covar(1:mfit,1:mfit)
    beta(1:mfit)=da(1:mfit,1)
    a=atry
else                                                  Failure, increase alamda and return.
    alamda=10.0_sp*alamda
    chisq=ochisq
end if
END SUBROUTINE mrqmin_private

SUBROUTINE mrqcof(a,alpha,beta)
REAL(SP), DIMENSION(:), INTENT(IN) :: a
REAL(SP), DIMENSION(:), INTENT(OUT) :: beta
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: alpha
    Used by mrqmin to evaluate the linearized fitting matrix alpha, and vector beta as in
    (15.5.8), and calculate χ².
INTEGER(I4B) :: j,k,l,m
REAL(SP), DIMENSION(size(x),size(a)) :: dyda
REAL(SP), DIMENSION(size(x)) :: dy,sig2i,wt,ymod
call funcs(x,a,ymod,dyda)                             Loop over all the data.
sig2i=1.0_sp/(sig**2)
dy=y-ymod
j=0
do l=1,ma
    if (maska(l)) then
        j=j+1
        wt=dyda(:,l)*sig2i
        k=0
        do m=1,l
            if (maska(m)) then
                k=k+1
                alpha(j,k)=dot_product(wt,dyda(:,m))
                alpha(k,j)=alpha(j,k)                 Fill in the symmetric side.
            end if
        end do
        beta(j)=dot_product(dy,wt)
    end if
end do
chisq=dot_product(dy**2,sig2i)                        Find χ².
END SUBROUTINE mrqcof
END SUBROUTINE mrqmin
```

f90 The organization of this routine is similar to that of amoeba, discussed on p. 1209. We want to keep the argument list of mrqcof to a minimum, but we want to make clear what global variables it accesses, and protect mrqmin_private's name space.

REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: atry,beta These arrays, as well as da, are allocated with the correct dimensions on the first call to mrqmin.

They need to retain their values between calls, so they are declared with the SAVE attribute. They get deallocated only on the final call when alamda=0.

`call diagmult(...)` See discussion of diagadd after hqr on p. 1234.

`atry=a+unpack(da(1:mfit,1),maska,0.0_sp)` maska controls which elements of a get incremented by da and which by 0.

⋆ ⋆ ⋆

```
SUBROUTINE fgauss(x,a,y,dyda)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
```

$y(x;a)$ is the sum of $N/3$ Gaussians (15.5.16). Here N is the length of the vectors x, y and a, while dyda is an $N \times N$ matrix. The amplitude, center, and width of the Gaussians are stored in consecutive locations of a: a(i) $= B_k$, a(i+1) $= E_k$, a(i+2) $= G_k$, $k = 1, \ldots, N/3$.

```
INTEGER(I4B) :: i,na,nx
REAL(SP), DIMENSION(size(x)) :: arg,ex,fac
nx=assert_eq(size(x),size(y),size(dyda,1),'fgauss: nx')
na=assert_eq(size(a),size(dyda,2),'fgauss: na')
y(:)=0.0
do i=1,na-1,3
    arg(:)=(x(:)-a(i+1))/a(i+2)
    ex(:)=exp(-arg(:)**2)
    fac(:)=a(i)*ex(:)*2.0_sp*arg(:)
    y(:)=y(:)+a(i)*ex(:)
    dyda(:,i)=ex(:)
    dyda(:,i+1)=fac(:)/a(i+2)
    dyda(:,i+2)=fac(:)*arg(:)/a(i+2)
end do
END SUBROUTINE fgauss
```

⋆ ⋆ ⋆

```
SUBROUTINE medfit(x,y,a,b,abdev)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : select
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: a,b,abdev
```

Fits $y = a + bx$ by the criterion of least absolute deviations. The same-size arrays x and y are the input experimental points. The fitted parameters a and b are output, along with abdev, which is the mean absolute deviation (in y) of the experimental points from the fitted line.

```
INTEGER(I4B) :: ndata
REAL(SP) :: aa
call medfit_private
CONTAINS
SUBROUTINE medfit_private
IMPLICIT NONE
REAL(SP) :: b1,b2,bb,chisq,del,f,f1,f2,sigb,sx,sxx,sxy,sy
REAL(SP), DIMENSION(size(x)) :: tmp
ndata=assert_eq(size(x),size(y),'medfit')
sx=sum(x)
sy=sum(y)
sxy=dot_product(x,y)
```

As a first guess for a and b, we will find the least squares fitting line.

```
sxx=dot_product(x,x)
del=ndata*sxx-sx**2
aa=(sxx*sy-sx*sxy)/del                  Least squares solutions.
bb=(ndata*sxy-sx*sy)/del
tmp(:)=y(:)-(aa+bb*x(:))
chisq=dot_product(tmp,tmp)
sigb=sqrt(chisq/del)                    The standard deviation will give some idea of how
b1=bb                                       big an iteration step to take.
f1=rofunc(b1)
b2=bb+sign(3.0_sp*sigb,f1)              Guess bracket as 3-σ away, in the downhill direction
f2=rofunc(b2)                               known from f1.
if (b2 == b1) then
    a=aa
    b=bb
    RETURN
endif
do                                      Bracketing.
    if (f1*f2 <= 0.0) exit
    bb=b2+1.6_sp*(b2-b1)
    b1=b2
    f1=f2
    b2=bb
    f2=rofunc(b2)
end do
sigb=0.01_sp*sigb                       Refine until error a negligible number of standard de-
do                                          viations.
    if (abs(b2-b1) <= sigb) exit
    bb=b1+0.5_sp*(b2-b1)                Bisection.
    if (bb == b1 .or. bb == b2) exit
    f=rofunc(bb)
    if (f*f1 >= 0.0) then
        f1=f
        b1=bb
    else
        f2=f
        b2=bb
    end if
end do
a=aa
b=bb
abdev=abdev/ndata
END SUBROUTINE medfit_private

FUNCTION rofunc(b)
IMPLICIT NONE
REAL(SP), INTENT(IN) :: b
REAL(SP) :: rofunc
REAL(SP), PARAMETER :: EPS=epsilon(b)
    Evaluates the right-hand side of equation (15.7.16) for a given value of b.
INTEGER(I4B) :: j
REAL(SP), DIMENSION(size(x)) :: arr,d
arr(:)=y(:)-b*x(:)
if (mod(ndata,2) == 0) then
    j=ndata/2
    aa=0.5_sp*(select(j,arr)+select(j+1,arr))
else
    aa=select((ndata+1)/2,arr)
end if
d(:)=y(:)-(b*x(:)+aa)
abdev=sum(abs(d))
where (y(:) /= 0.0) d(:)=d(:)/abs(y(:))
rofunc=sum(x(:)*sign(1.0_sp,d(:)), mask=(abs(d(:)) > EPS) )
END FUNCTION rofunc
END SUBROUTINE medfit
```

f90 The organization of this routine is similar to that of `amoeba` discussed on p. 1209. We want to keep the argument list of `rofunc` to a minimum, but we want to make clear what global variables it accesses and protect `medfit_private`'s name space. In the Fortran 77 version, we kept the only argument as `b` by passing the global variables in a common block. This required us to make copies of the arrays `x` and `y`. An alternative Fortran 90 implementation would be to use a module with pointers to the arguments of `medfit` like `x` and `y` that need to be passed to `rofunc`. We think the `medfit_private` construction is simpler.

Chapter B16. Integration of Ordinary Differential Equations

```
SUBROUTINE rk4(y,dydx,x,h,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
```

Given values for the N variables `y` and their derivatives `dydx` known at `x`, use the fourth-order Runge-Kutta method to advance the solution over an interval `h` and return the incremented variables as `yout`, which need not be a distinct array from `y`. `y`, `dydx` and `yout` are all of length N. The user supplies the subroutine `derivs(x,y,dydx)`, which returns derivatives `dydx` at `x`.

```
INTEGER(I4B) :: ndum
REAL(SP) :: h6,hh,xh
REAL(SP), DIMENSION(size(y)) :: dym,dyt,yt
ndum=assert_eq(size(y),size(dydx),size(yout),'rk4')
hh=h*0.5_sp
h6=h/6.0_sp
xh=x+hh
yt=y+hh*dydx                          First step.
call derivs(xh,yt,dyt)                Second step.
yt=y+hh*dyt
call derivs(xh,yt,dym)                Third step.
yt=y+h*dym
dym=dyt+dym
call derivs(x+h,yt,dyt)               Fourth step.
yout=y+h6*(dydx+dyt+2.0_sp*dym)       Accumulate increments with proper weights.
END SUBROUTINE rk4
```

⋆ ⋆ ⋆

```
MODULE rkdumb_path                    Storage of results.
USE nrtype
REAL(SP), DIMENSION(:), ALLOCATABLE:: xx
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: y
END MODULE rkdumb_path
```

```
SUBROUTINE rkdumb(vstart,x1,x2,nstep,derivs)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : rk4
USE rkdumb_path
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: vstart
REAL(SP), INTENT(IN) :: x1,x2
INTEGER(I4B), INTENT(IN) :: nstep
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
```

Starting from N initial values `vstart` known at `x1`, use fourth-order Runge-Kutta to advance `nstep` equal increments to `x2`. The user-supplied subroutine `derivs(x,y,dydx)` evaluates derivatives. Results are stored in the module variables `xx` and `y`.

```
INTEGER(I4B) :: k
REAL(SP) :: h,x
REAL(SP), DIMENSION(size(vstart)) :: dv,v
v(:)=vstart(:)                                  Load starting values.
if (allocated(xx)) deallocate(xx)               Clear out old stored variables if necessary.
if (allocated(y)) deallocate(y)
allocate(xx(nstep+1))                           Allocate storage for saved values.
allocate(y(size(vstart),nstep+1))
y(:,1)=v(:)
xx(1)=x1
x=x1
h=(x2-x1)/nstep
do k=1,nstep                                    Take nstep steps.
    call derivs(x,v,dv)
    call rk4(v,dv,x,h,v,derivs)
    if (x+h == x) call nrerror('stepsize not significant in rkdumb')
    x=x+h
    xx(k+1)=x                                   Store intermediate steps.
    y(:,k+1)=v(:)
end do
END SUBROUTINE rkdumb
```

f90 `MODULE rkdumb_path` This routine needs straightforward communication of arrays with the calling program. The dimension of the arrays is not known in advance, and if the routine is called a second time we need to throw away the old array information. The Fortran 90 construction for this is to declare allocatable arrays in a module, and then test them at the beginning of the routine with `if (allocated...)`.

⋆ ⋆ ⋆

```
SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : rkck
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
```

```
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
```

Fifth order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. `y`, `dydx`, and `yscal` are all of the same length. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is the user-supplied subroutine that computes the right-hand-side derivatives.

```
INTEGER(I4B) :: ndum
REAL(SP) :: errmax,h,htemp,xnew
REAL(SP), DIMENSION(size(y)) :: yerr,ytemp
REAL(SP), PARAMETER :: SAFETY=0.9_sp,PGROW=-0.2_sp,PSHRNK=-0.25_sp,&
    ERRCON=1.89e-4
```

The value ERRCON equals `(5/SAFETY)**(1/PGROW)`, see use below.

```
ndum=assert_eq(size(y),size(dydx),size(yscal),'rkqs')
h=htry                                      Set stepsize to the initial trial value.
do
    call rkck(y,dydx,x,h,ytemp,yerr,derivs)      Take a step.
    errmax=maxval(abs(yerr(:)/yscal(:)))/eps     Evaluate accuracy.
    if (errmax <= 1.0) exit                 Step succeeded.
    htemp=SAFETY*h*(errmax**PSHRNK)         Truncation error too large, reduce stepsize.
    h=sign(max(abs(htemp),0.1_sp*abs(h)),h)      No more than a factor of 10.
    xnew=x+h
    if (xnew == x) call nrerror('stepsize underflow in rkqs')
end do                                      Go back for another try.
if (errmax > ERRCON) then                   Compute size of next step.
    hnext=SAFETY*h*(errmax**PGROW)
else                                        No more than a factor of 5 increase.
    hnext=5.0_sp*h
end if
hdid=h
x=x+h
y(:)=ytemp(:)
END SUBROUTINE rkqs
```

$\star \quad \star \quad \star$

```
SUBROUTINE rkck(y,dydx,x,h,yout,yerr,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), INTENT(IN) :: x,h
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout,yerr
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
```

Given values for N variables `y` and their derivatives `dydx` known at `x`, use the fifth order Cash-Karp Runge-Kutta method to advance the solution over an interval `h` and return

the incremented variables as yout. Also return an estimate of the local truncation error in yout using the embedded fourth order method. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

```
INTEGER(I4B) :: ndum
REAL(SP), DIMENSION(size(y)) :: ak2,ak3,ak4,ak5,ak6,ytemp
REAL(SP), PARAMETER :: A2=0.2_sp,A3=0.3_sp,A4=0.6_sp,A5=1.0_sp,&
    A6=0.875_sp,B21=0.2_sp,B31=3.0_sp/40.0_sp,B32=9.0_sp/40.0_sp,&
    B41=0.3_sp,B42=-0.9_sp,B43=1.2_sp,B51=-11.0_sp/54.0_sp,&
    B52=2.5_sp,B53=-70.0_sp/27.0_sp,B54=35.0_sp/27.0_sp,&
    B61=1631.0_sp/55296.0_sp,B62=175.0_sp/512.0_sp,&
    B63=575.0_sp/13824.0_sp,B64=44275.0_sp/110592.0_sp,&
    B65=253.0_sp/4096.0_sp,C1=37.0_sp/378.0_sp,&
    C3=250.0_sp/621.0_sp,C4=125.0_sp/594.0_sp,&
    C6=512.0_sp/1771.0_sp,DC1=C1-2825.0_sp/27648.0_sp,&
    DC3=C3-18575.0_sp/48384.0_sp,DC4=C4-13525.0_sp/55296.0_sp,&
    DC5=-277.0_sp/14336.0_sp,DC6=C6-0.25_sp
ndum=assert_eq(size(y),size(dydx),size(yout),size(yerr),'rkck')
ytemp=y+B21*h*dydx                          First step.
call derivs(x+A2*h,ytemp,ak2)               Second step.
ytemp=y+h*(B31*dydx+B32*ak2)
call derivs(x+A3*h,ytemp,ak3)               Third step.
ytemp=y+h*(B41*dydx+B42*ak2+B43*ak3)
call derivs(x+A4*h,ytemp,ak4)               Fourth step.
ytemp=y+h*(B51*dydx+B52*ak2+B53*ak3+B54*ak4)
call derivs(x+A5*h,ytemp,ak5)               Fifth step.
ytemp=y+h*(B61*dydx+B62*ak2+B63*ak3+B64*ak4+B65*ak5)
call derivs(x+A6*h,ytemp,ak6)               Sixth step.
yout=y+h*(C1*dydx+C3*ak3+C4*ak4+C6*ak6)     Accumulate increments with proper weights.
yerr=h*(DC1*dydx+DC3*ak3+DC4*ak4+DC5*ak5+DC6*ak6)
```

Estimate error as difference between fourth and fifth order methods.

```
END SUBROUTINE rkck
```

⋆ ⋆ ⋆

```
MODULE ode_path
USE nrtype
INTEGER(I4B) :: nok,nbad,kount
LOGICAL(LGT), SAVE :: save_steps=.false.
REAL(SP) :: dxsav
REAL(SP), DIMENSION(:), POINTER :: xp
REAL(SP), DIMENSION(:,:), POINTER :: yp
END MODULE ode_path
```

On output nok and nbad are the number of good and bad (but retried and fixed) steps taken. If save_steps is set to true in the calling program, then intermediate values are stored in xp and yp at intervals greater than dxsav. kount is the total number of saved steps.

```
SUBROUTINE odeint(ystart,x1,x2,eps,h1,hmin,derivs,rkqs)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE ode_path
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: ystart
REAL(SP), INTENT(IN) :: x1,x2,eps,h1,hmin
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs

    SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
    USE nrtype
```

```
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(OUT) :: hdid,hnext
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE rkqs
END INTERFACE
REAL(SP), PARAMETER :: TINY=1.0e-30_sp
INTEGER(I4B), PARAMETER :: MAXSTP=10000
```

Runge-Kutta driver with adaptive stepsize control. Integrate the array of starting values `ystart` from `x1` to `x2` with accuracy `eps`, storing intermediate results in the module variables in `ode_path`. `h1` should be set as a guessed first stepsize, `hmin` as the minimum allowed stepsize (can be zero). On output `ystart` is replaced by values at the end of the integration interval. `derivs` is the user-supplied subroutine for calculating the right-hand-side derivative, while `rkqs` is the name of the stepper routine to be used.

```
INTEGER(I4B) :: nstp
REAL(SP) :: h,hdid,hnext,x,xsav
REAL(SP), DIMENSION(size(ystart)) :: dydx,y,yscal
x=x1
h=sign(h1,x2-x1)
nok=0
nbad=0
kount=0
y(:)=ystart(:)
if (save_steps) then
    xsav=x-2.0_sp*dxsav                          Assures storage of first step.
    nullify(xp,yp)                               Pointers nullified here, but memory not
    allocate(xp(256))                                deallocated. If odeint is called mul-
    allocate(yp(size(ystart),size(xp)))              tiple times, calling program should
end if                                               deallocate xp and yp between calls.
do nstp=1,MAXSTP                                 Take at most MAXSTP steps.
    call derivs(x,y,dydx)
    yscal(:)=abs(y(:))+abs(h*dydx(:))+TINY
```

Scaling used to monitor accuracy. This general purpose choice can be modified if need be.

```
    if (save_steps .and. (abs(x-xsav) > abs(dxsav))) &      Store intermediate results.
        call save_a_step
    if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x          If stepsize can overshoot, decrease.
    call rkqs(y,dydx,x,h,eps,yscal,hdid,hnext,derivs)
    if (hdid == h) then
        nok=nok+1
    else
        nbad=nbad+1
    end if
    if ((x-x2)*(x2-x1) >= 0.0) then              Are we done?
        ystart(:)=y(:)
        if (save_steps) call save_a_step         Save final step.
        RETURN                                   Normal exit.
    end if
    if (abs(hnext) < hmin)&
        call nrerror('stepsize smaller than minimum in odeint')
    h=hnext
end do
call nrerror('too many steps in odeint')
```

```
CONTAINS
SUBROUTINE save_a_step
kount=kount+1
if (kount > size(xp)) then
    xp=>reallocate(xp,2*size(xp))
    yp=>reallocate(yp,size(yp,1),size(xp))
end if
xp(kount)=x
yp(:,kount)=y(:)
xsav=x
END SUBROUTINE save_a_step
END SUBROUTINE odeint
```

f90 MODULE ode_path The situation here is similar to rkdumb_path, except we don't know at run time how much storage to allocate. We may need to use reallocate from nrutil to increase the storage. The solution is pointers to arrays, with a nullify to be sure the pointer status is well-defined at the beginning of the routine.

SUBROUTINE save_a_step An internal subprogram with no arguments is like a macro in C: you could imagine just copying its code wherever it is called in the parent routine.

⋆ ⋆ ⋆

```
SUBROUTINE mmid(y,dydx,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), INTENT(IN) :: xs,htot
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
```

Modified midpoint step. Dependent variable vector y and its derivative vector dydx are input at xs. Also input is htot, the total step to be taken, and nstep, the number of substeps to be used. The output is returned as yout, which need not be a distinct array from y; if it is distinct, however, then y and dydx are returned undamaged. y, dydx, and yout must all have the same length.

```
INTEGER(I4B) :: n,ndum
REAL(SP) :: h,h2,x
REAL(SP), DIMENSION(size(y)) :: ym,yn
ndum=assert_eq(size(y),size(dydx),size(yout),'mmid')
h=htot/nstep                          Stepsize this trip.
ym=y
yn=y+h*dydx                           First step.
x=xs+h
call derivs(x,yn,yout)                Will use yout for temporary storage of derivatives.
h2=2.0_sp*h
do n=2,nstep                          General step.
    call swap(ym,yn)
    yn=yn+h2*yout
```

```
    x=x+h
    call derivs(x,yn,yout)
end do
yout=0.5_sp*(ym+yn+h*yout)          Last step.
END SUBROUTINE mmid
```

⋆ ⋆ ⋆

```
SUBROUTINE bsstep(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,cumsum,iminloc,nrerror,&
    outerdiff,outerprod,upper_triangle
USE nr, ONLY : mmid,pzextr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
INTEGER(I4B), PARAMETER :: IMAX=9, KMAXX=IMAX-1
REAL(SP), PARAMETER :: SAFE1=0.25_sp,SAFE2=0.7_sp,REDMAX=1.0e-5_sp,&
    REDMIN=0.7_sp,TINY=1.0e-30_sp,SCALMX=0.1_sp
```

Bulirsch-Stoer step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is the user-supplied subroutine that computes the right-hand-side derivatives. `y`, `dydx`, and `yscal` must all have the same length. Be sure to set `htry` on successive steps to the value of `hnext` returned from the previous step, as is the case if the routine is called by `odeint`.

Parameters: `KMAXX` is the maximum row number used in the extrapolation; `IMAX` is the next row number; `SAFE1` and `SAFE2` are safety factors; `REDMAX` is the maximum factor used when a stepsize is reduced, `REDMIN` the minimum; `TINY` prevents division by zero; $1/$`SCALMX` is the maximum factor by which a stepsize can be increased.

```
INTEGER(I4B) :: k,km,ndum
INTEGER(I4B), DIMENSION(IMAX) :: nseq = (/ 2,4,6,8,10,12,14,16,18 /)
INTEGER(I4B), SAVE :: kopt,kmax
REAL(SP), DIMENSION(KMAXX,KMAXX), SAVE :: alf
REAL(SP), DIMENSION(KMAXX) :: err
REAL(SP), DIMENSION(IMAX), SAVE :: a
REAL(SP), SAVE :: epsold = -1.0_sp,xnew
REAL(SP) :: eps1,errmax,fact,h,red,scale,wrkmin,xest
REAL(SP), DIMENSION(size(y)) :: yerr,ysav,yseq
LOGICAL(LGT) :: reduct
LOGICAL(LGT), SAVE :: first=.true.
ndum=assert_eq(size(y),size(dydx),size(yscal),'bsstep')
if (eps /= epsold) then                    A new tolerance, so reinitialize.
    hnext=-1.0e29_sp                       "Impossible" values.
    xnew=-1.0e29_sp
    eps1=SAFE1*eps
    a(:)=cumsum(nseq,1)
      Compute \alpha(k,q):
    where (upper_triangle(KMAXX,KMAXX)) alf=eps1** &
        (outerdiff(a(2:),a(2:))/outerprod(arth( &
```

```
        3.0_sp,2.0_sp,KMAXX),(a(2:)-a(1)+1.0_sp)))
    epsold=eps
    do kopt=2,KMAXX-1                          Determine optimal row number for con-
        if (a(kopt+1) > a(kopt)*alf(kopt-1,kopt)) exit      vergence.
    end do
    kmax=kopt
end if
h=htry
ysav(:)=y(:)                                   Save the starting values.
if (h /= hnext .or. x /= xnew) then            A new stepsize or a new integration: Re-
    first=.true.                                   establish the order window.
    kopt=kmax
end if
reduct=.false.
main_loop: do
    do k=1,kmax                                Evaluate the sequence of modified mid-
        xnew=x+h                                   point integrations.
        if (xnew == x) call nrerror('step size underflow in bsstep')
        call mmid(ysav,dydx,x,h,nseq(k),yseq,derivs)
        xest=(h/nseq(k))**2                    Squared, since error series is even.
        call pzextr(k,xest,yseq,y,yerr)        Perform extrapolation.
        if (k /= 1) then                       Compute normalized error estimate \epsilon(k).
            errmax=maxval(abs(yerr(:)/yscal(:)))
            errmax=max(TINY,errmax)/eps        Scale error relative to tolerance.
            km=k-1
            err(km)=(errmax/SAFE1)**(1.0_sp/(2*km+1))
        end if
        if (k /= 1 .and. (k >= kopt-1 .or. first)) then     In order window.
            if (errmax < 1.0) exit main_loop                Converged.
            if (k == kmax .or. k == kopt+1) then            Check for possible step-
                red=SAFE2/err(km)                               size reduction.
                exit
            else if (k == kopt) then
                if (alf(kopt-1,kopt) < err(km)) then
                    red=1.0_sp/err(km)
                    exit
                end if
            else if (kopt == kmax) then
                if (alf(km,kmax-1) < err(km)) then
                    red=alf(km,kmax-1)*SAFE2/err(km)
                    exit
                end if
            else if (alf(km,kopt) < err(km)) then
                red=alf(km,kopt-1)/err(km)
                exit
            end if
        end if
    end do
    red=max(min(red,REDMIN),REDMAX)            Reduce stepsize by at least REDMIN and
    h=h*red                                        at most REDMAX.
    reduct=.true.
end do main_loop                               Try again.
x=xnew                                         Successful step taken.
hdid=h
first=.false.
kopt=1+iminloc(a(2:km+1)*max(err(1:km),SCALMX))
  Compute optimal row for convergence and corresponding stepsize.
scale=max(err(kopt-1),SCALMX)
wrkmin=scale*a(kopt)
hnext=h/scale
if (kopt >= k .and. kopt /= kmax .and. .not. reduct) then     Check for possible or-
    fact=max(scale/alf(kopt-1,kopt),SCALMX)                       der increase, but
    if (a(kopt+1)*fact <= wrkmin) then                            not if stepsize was
        hnext=h/fact                                              just reduced.
```

```
        kopt=kopt+1
    end if
end if
END SUBROUTINE bsstep
```

`a(:)=cumsum(nseq,1)` The function `cumsum` in `nrutil` with the optional argument `seed=1` gives a direct implementation of equation (16.4.6).

`where (upper_triangle(KMAXX,KMAXX))...` The `upper_triangle` function in `nrutil` returns an upper triangular logical mask. As used here, the mask is true everywhere in the upper triangle of a `KMAXX` $\times$ `KMAXX` matrix, excluding the diagonal. An optional integer argument `extra` allows additional diagonals to be set to true. With `extra=1` the upper triangle including the diagonal would be true.

`main_loop: do` Using a named do-loop provides clear structured code that required `goto`'s in the Fortran 77 version.

`kopt=1+iminloc(...)` See the discussion of `imaxloc` on p. 1017.

$\star$ $\star$ $\star$

```
SUBROUTINE pzextr(iest,xest,yest,yz,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iest
REAL(SP), INTENT(IN) :: xest
REAL(SP), DIMENSION(:), INTENT(IN) :: yest
REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
```

Use polynomial extrapolation to evaluate N functions at $x = 0$ by fitting a polynomial to a sequence of estimates with progressively smaller values $x =$ `xest`, and corresponding function vectors `yest`. This call is number `iest` in the sequence of calls. Extrapolated function values are output as `yz`, and their estimated error is output as `dy`. `yest`, `yz`, and `dy` are arrays of length N.

```
INTEGER(I4B), PARAMETER :: IEST_MAX=16
INTEGER(I4B) :: j,nv
INTEGER(I4B), SAVE :: nvold=-1
REAL(SP) :: delta,f1,f2
REAL(SP), DIMENSION(size(yz)) :: d,tmp,q
REAL(SP), DIMENSION(IEST_MAX), SAVE :: x
REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: qcol
nv=assert_eq(size(yz),size(yest),size(dy),'pzextr')
if (iest > IEST_MAX) call &
    nrerror('pzextr: probable misuse, too much extrapolation')
if (nv /= nvold) then                Set up internal storage.
    if (allocated(qcol)) deallocate(qcol)
    allocate(qcol(nv,IEST_MAX))
    nvold=nv
end if
x(iest)=xest                         Save current independent variable.
dy(:)=yest(:)
yz(:)=yest(:)
if (iest == 1) then                  Store first estimate in first column.
    qcol(:,1)=yest(:)
else
    d(:)=yest(:)
    do j=1,iest-1
        delta=1.0_sp/(x(iest-j)-xest)
        f1=xest*delta
        f2=x(iest-j)*delta
        q(:)=qcol(:,j)               Propagate tableau 1 diagonal more.
```

```
        qcol(:,j)=dy(:)
        tmp(:)=d(:)-q(:)
        dy(:)=f1*tmp(:)
        d(:)=f2*tmp(:)
        yz(:)=yz(:)+dy(:)
    end do
    qcol(:,iest)=dy(:)
end if
END SUBROUTINE pzextr
```

f90 `REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: qcol` The second dimension of `qcol` is known at compile time to be `IEST_MAX`, but the first dimension is known only at run time, from `size(yz)`. The language requires us to have all dimensions allocatable if any one of them is.

`if (nv /= nvold) then...` This routine generally gets called many times with `iest` cycling repeatedly through the values $1, 2, \ldots,$ up to some value less than `IEST_MAX`. The number of variables, `nv`, is fixed during the solution of the problem. The routine might be called again in solving a different problem with a new value of `nv`. This if block ensures that `qcol` is dimensioned correctly both for the first and subsequent problems, if any.

```
SUBROUTINE rzextr(iest,xest,yest,yz,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iest
REAL(SP), INTENT(IN) :: xest
REAL(SP), DIMENSION(:), INTENT(IN) :: yest
REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
    Exact substitute for pzextr, but uses diagonal rational function extrapolation instead of
    polynomial extrapolation.
INTEGER(I4B), PARAMETER :: IEST_MAX=16
INTEGER(I4B) :: k,nv
INTEGER(I4B), SAVE :: nvold=-1
REAL(SP), DIMENSION(size(yz)) :: yy,v,c,b,b1,ddy
REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: d
REAL(SP), DIMENSION(IEST_MAX), SAVE :: fx,x
nv=assert_eq(size(yz),size(dy),size(yest),'rzextr')
if (iest > IEST_MAX) call &
    nrerror('rzextr: probable misuse, too much extrapolation')
if (nv /= nvold) then
    if (allocated(d)) deallocate(d)
    allocate(d(nv,IEST_MAX))
    nvold=nv
end if
x(iest)=xest                    Save current independent variable.
if (iest == 1) then
    yz=yest
    d(:,1)=yest
    dy=yest
else
    fx(2:iest)=x(iest-1:1:-1)/xest
    yy=yest                     Evaluate next diagonal in tableau.
    v=d(1:nv,1)
    c=yy
    d(1:nv,1)=yy
    do k=2,iest
        b1=fx(k)*v
        b=b1-c
        where (b /= 0.0)
```

```
            b=(c-v)/b
            ddy=c*b
            c=b1*b
        elsewhere                Care needed to avoid division by 0.
            ddy=v
        end where
        if (k /= iest) v=d(1:nv,k)
        d(1:nv,k)=ddy
        yy=yy+ddy
    end do
    dy=ddy
    yz=yy
end if
END SUBROUTINE rzextr
```

⋆ ⋆ ⋆

```
SUBROUTINE stoerm(y,d2y,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: y,d2y
REAL(SP), INTENT(IN) :: xs,htot
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
```

Stoermer's rule for integrating $y'' = f(x,y)$ for a system of n equations. On input y contains y in its first n elements and y' in its second n elements, all evaluated at xs. d2y contains the right-hand-side function f (also evaluated at xs) in its first n elements. Its second n elements are not referenced. Also input is htot, the total step to be taken, and nstep, the number of substeps to be used. The output is returned as yout, with the same storage arrangement as y. derivs is the user-supplied subroutine that calculates f.

```
INTEGER(I4B) :: neqn,neqn1,nn,nv
REAL(SP) :: h,h2,halfh,x
REAL(SP), DIMENSION(size(y)) :: ytemp
nv=assert_eq(size(y),size(d2y),size(yout),'stoerm')
neqn=nv/2                                   Number of equations.
neqn1=neqn+1
h=htot/nstep                                Stepsize this trip.
halfh=0.5_sp*h                              First step.
ytemp(neqn1:nv)=h*(y(neqn1:nv)+halfh*d2y(1:neqn))
ytemp(1:neqn)=y(1:neqn)+ytemp(neqn1:nv)
x=xs+h
call derivs(x,ytemp,yout)                   Use yout for temporary storage of deriva-
h2=h*h                                          tives.
do nn=2,nstep                               General step.
    ytemp(neqn1:nv)=ytemp(neqn1:nv)+h2*yout(1:neqn)
    ytemp(1:neqn)=ytemp(1:neqn)+ytemp(neqn1:nv)
    x=x+h
    call derivs(x,ytemp,yout)
end do
yout(neqn1:nv)=ytemp(neqn1:nv)/h+halfh*yout(1:neqn)     Last step.
yout(1:neqn)=ytemp(1:neqn)
END SUBROUTINE stoerm
```

⋆ ⋆ ⋆

```
SUBROUTINE stiff(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,diagadd,nrerror
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs

    SUBROUTINE jacobn(x,y,dfdx,dfdy)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
    END SUBROUTINE jacobn
END INTERFACE
INTEGER(I4B), PARAMETER :: MAXTRY=40
REAL(SP), PARAMETER :: SAFETY=0.9_sp,GROW=1.5_sp,PGROW=-0.25_sp,&
    SHRNK=0.5_sp,PSHRNK=-1.0_sp/3.0_sp,ERRCON=0.1296_sp,&
    GAM=1.0_sp/2.0_sp,&
    A21=2.0_sp,A31=48.0_sp/25.0_sp,A32=6.0_sp/25.0_sp,C21=-8.0_sp,&
    C31=372.0_sp/25.0_sp,C32=12.0_sp/5.0_sp,&
    C41=-112.0_sp/125.0_sp,C42=-54.0_sp/125.0_sp,&
    C43=-2.0_sp/5.0_sp,B1=19.0_sp/9.0_sp,B2=1.0_sp/2.0_sp,&
    B3=25.0_sp/108.0_sp,B4=125.0_sp/108.0_sp,E1=17.0_sp/54.0_sp,&
    E2=7.0_sp/36.0_sp,E3=0.0_sp,E4=125.0_sp/108.0_sp,&
    C1X=1.0_sp/2.0_sp,C2X=-3.0_sp/2.0_sp,C3X=121.0_sp/50.0_sp,&
    C4X=29.0_sp/250.0_sp,A2X=1.0_sp,A3X=3.0_sp/5.0_sp
```

Fourth order Rosenbrock step for integrating stiff ODEs, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied subroutine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. `y`, `dydx`, and `yscal` must have the same length.

Parameters: `GROW` and `SHRNK` are the largest and smallest factors by which stepsize can change in one step; `ERRCON=(GROW/SAFETY)**(1/PGROW)` and handles the case when `errmax` $\simeq 0$.

```
INTEGER(I4B) :: jtry,ndum
INTEGER(I4B), DIMENSION(size(y)) :: indx
REAL(SP), DIMENSION(size(y)) :: dfdx,dytmp,err,g1,g2,g3,g4,ysav
REAL(SP), DIMENSION(size(y),size(y)) :: a,dfdy
REAL(SP) :: d,errmax,h,xsav
ndum=assert_eq(size(y),size(dydx),size(yscal),'stiff')
xsav=x                                          Save initial values.
ysav(:)=y(:)
call jacobn(xsav,ysav,dfdx,dfdy)
```

The user must supply this subroutine to return the `n` $\times$ `n` matrix `dfdy` and the vector `dfdx`.

```
h=htry                                          Set stepsize to the initial trial value.
do jtry=1,MAXTRY
```

```
    a(:,:)=-dfdy(:,:)                    Set up the matrix 1 − γhf′.
    call diagadd(a,1.0_sp/(GAM*h))
    call ludcmp(a,indx,d)                LU decomposition of the matrix.
    g1=dydx+h*C1X*dfdx                   Set up right-hand side for g1.
    call lubksb(a,indx,g1)               Solve for g1.
    y=ysav+A21*g1                        Compute intermediate values of y and x.
    x=xsav+A2X*h
    call derivs(x,y,dytmp)               Compute dydx at the intermediate values.
    g2=dytmp+h*C2X*dfdx+C21*g1/h         Set up right-hand side for g2.
    call lubksb(a,indx,g2)               Solve for g2.
    y=ysav+A31*g1+A32*g2                 Compute intermediate values of y and x.
    x=xsav+A3X*h
    call derivs(x,y,dytmp)               Compute dydx at the intermediate values.
    g3=dytmp+h*C3X*dfdx+(C31*g1+C32*g2)/h        Set up right-hand side for g3.
    call lubksb(a,indx,g3)                       Solve for g3.
    g4=dytmp+h*C4X*dfdx+(C41*g1+C42*g2+C43*g3)/h Set up right-hand side for g4.
    call lubksb(a,indx,g4)                       Solve for g4.
    y=ysav+B1*g1+B2*g2+B3*g3+B4*g4       Get fourth order estimate of y and error es-
    err=E1*g1+E2*g2+E3*g3+E4*g4              timate.
    x=xsav+h
    if (x == xsav) call &
        nrerror('stepsize not significant in stiff')
    errmax=maxval(abs(err/yscal))/eps    Evaluate accuracy.
    if (errmax <= 1.0) then              Step succeeded. Compute size of next step
        hdid=h                               and return.
        hnext=merge(SAFETY*h*errmax**PGROW, GROW*h, &
            errmax > ERRCON)
        RETURN
    else                                 Truncation error too large, reduce stepsize.
        hnext=SAFETY*h*errmax**PSHRNK
        h=sign(max(abs(hnext),SHRNK*abs(h)),h)
    end if
end do                                   Go back and retry step.
call nrerror('exceeded MAXTRY in stiff')
END SUBROUTINE stiff
```

f90 `call diagadd(...)` See discussion of diagadd after hqr on p. 1234.

```
SUBROUTINE jacobn(x,y,dfdx,dfdy)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
   Routine for Jacobi matrix corresponding to example in equations (16.6.27).
dfdx(:)=0.0
dfdy(1,1)=-0.013_sp-1000.0_sp*y(3)
dfdy(1,2)=0.0
dfdy(1,3)=-1000.0_sp*y(1)
dfdy(2,1)=0.0
dfdy(2,2)=-2500.0_sp*y(3)
dfdy(2,3)=-2500.0_sp*y(2)
dfdy(3,1)=-0.013_sp-1000.0_sp*y(3)
dfdy(3,2)=-2500.0_sp*y(3)
dfdy(3,3)=-1000.0_sp*y(1)-2500.0_sp*y(2)
END SUBROUTINE jacobn
```

```
SUBROUTINE derivs(x,y,dydx)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
```

Routine for right-hand side of example in equations (16.6.27).

```
dydx(1)=-0.013_sp*y(1)-1000.0_sp*y(1)*y(3)
dydx(2)=-2500.0_sp*y(2)*y(3)
dydx(3)=-0.013_sp*y(1)-1000.0_sp*y(1)*y(3)-2500.0_sp*y(2)*y(3)
END SUBROUTINE derivs
```

⋆ ⋆ ⋆

```
SUBROUTINE simpr(y,dydx,dfdx,dfdy,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,diagadd
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xs,htot
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx,dfdx
REAL(SP), DIMENSION(:,:), INTENT(IN) :: dfdy
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
END INTERFACE
```

Performs one step of semi-implicit midpoint rule. Input are the dependent variable y, its derivative dydx, the derivative of the right-hand side with respect to x, dfdx, which are all vectors of length N, and the $N \times N$ Jacobian dfdy at xs. Also input are htot, the total step to be taken, and nstep, the number of substeps to be used. The output is returned as yout, a vector of length N. derivs is the user-supplied subroutine that calculates dydx.

```
INTEGER(I4B) :: ndum,nn
INTEGER(I4B), DIMENSION(size(y)) :: indx
REAL(SP) :: d,h,x
REAL(SP), DIMENSION(size(y)) :: del,ytemp
REAL(SP), DIMENSION(size(y),size(y)) :: a
ndum=assert_eq((/size(y),size(dydx),size(dfdx),size(dfdy,1),&
    size(dfdy,2),size(yout)/),'simpr')
h=htot/nstep                        Stepsize this trip.
a(:,:)=-h*dfdy(:,:)                 Set up the matrix 1 − hf′.
call diagadd(a,1.0_sp)
call ludcmp(a,indx,d)               LU decomposition of the matrix.
yout=h*(dydx+h*dfdx)                Set up right-hand side for first step. Use yout for
call lubksb(a,indx,yout)                temporary storage.
del=yout                            First step.
ytemp=y+del
x=xs+h
call derivs(x,ytemp,yout)           Use yout for temporary storage of derivatives.
do nn=2,nstep                       General step.
    yout=h*yout-del                 Set up right-hand side for general step.
    call lubksb(a,indx,yout)
    del=del+2.0_sp*yout
    ytemp=ytemp+del
    x=x+h
    call derivs(x,ytemp,yout)
```

```
end do
yout=h*yout-del                          Set up right-hand side for last step.
call lubksb(a,indx,yout)
yout=ytemp+yout                          Take last step.
END SUBROUTINE simpr
```

f90 `call diagadd(...)` See discussion of diagadd after hqr on p. 1234.

⋆ ⋆ ⋆

```
SUBROUTINE stifbs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,cumsum,iminloc,nrerror,&
    outerdiff,outerprod,upper_triangle
USE nr, ONLY : simpr,pzextr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs

    SUBROUTINE jacobn(x,y,dfdx,dfdy)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
    END SUBROUTINE jacobn
END INTERFACE
INTEGER(I4B), PARAMETER :: IMAX=8, KMAXX=IMAX-1
REAL(SP), PARAMETER :: SAFE1=0.25_sp,SAFE2=0.7_sp,REDMAX=1.0e-5_sp,&
    REDMIN=0.7_sp,TINY=1.0e-30_sp,SCALMX=0.1_sp
```

Semi-implicit extrapolation step for integrating stiff ODEs, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y` and its derivative `dydx` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied subroutine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. `y`, `dydx`, and `yscal` must all have the same length. Be sure to set `htry` on successive steps to the value of `hnext` returned from the previous step, as is the case if the routine is called by `odeint`.

```
INTEGER(I4B) :: k,km,ndum
INTEGER(I4B), DIMENSION(IMAX) :: nseq = (/ 2,6,10,14,22,34,50,70 /)
```

Sequence is different from bsstep.

```
INTEGER(I4B), SAVE :: kopt,kmax,nvold=-1
REAL(SP), DIMENSION(KMAXX,KMAXX), SAVE :: alf
REAL(SP), DIMENSION(KMAXX) :: err
REAL(SP), DIMENSION(IMAX), SAVE :: a
REAL(SP), SAVE :: epsold = -1.0
REAL(SP) :: eps1,errmax,fact,h,red,scale,wrkmin,xest
```

```
REAL(SP), SAVE :: xnew
REAL(SP), DIMENSION(size(y)) :: dfdx,yerr,ysav,yseq
REAL(SP), DIMENSION(size(y),size(y)) :: dfdy
LOGICAL(LGT) :: reduct
LOGICAL(LGT), SAVE :: first=.true.
ndum=assert_eq(size(y),size(dydx),size(yscal),'stifbs')
if (eps /= epsold .or. nvold /= size(y)) then        Reinitialize also if number of vari-
    hnext=-1.0e29_sp                                      ables has changed.
    xnew=-1.0e29_sp
    eps1=SAFE1*eps
    a(:)=cumsum(nseq,1)
    where (upper_triangle(KMAXX,KMAXX)) alf=eps1** &
        (outerdiff(a(2:),a(2:))/outerprod(arth( &
        3.0_sp,2.0_sp,KMAXX),(a(2:)-a(1)+1.0_sp)))
    epsold=eps
    nvold=size(y)                                 Save number of variables.
    a(:)=cumsum(nseq,1+nvold)                     Add cost of Jacobian evaluations to work co-
    do kopt=2,KMAXX-1                                 efficients.
        if (a(kopt+1) > a(kopt)*alf(kopt-1,kopt)) exit
    end do
    kmax=kopt
end if
h=htry
ysav(:)=y(:)
call jacobn(x,y,dfdx,dfdy)                        Evaluate Jacobian.
if (h /= hnext .or. x /= xnew) then
    first=.true.
    kopt=kmax
end if
reduct=.false.
main_loop: do
    do k=1,kmax
        xnew=x+h
        if (xnew == x) call nrerror('step size underflow in stifbs')
        call simpr(ysav,dydx,dfdx,dfdy,x,h,nseq(k),yseq,derivs)
          Here is the call to the semi-implicit midpoint rule.
        xest=(h/nseq(k))**2                       The rest of the routine is identical to bsstep.
        call pzextr(k,xest,yseq,y,yerr)
        if (k /= 1) then
            errmax=maxval(abs(yerr(:)/yscal(:)))
            errmax=max(TINY,errmax)/eps
            km=k-1
            err(km)=(errmax/SAFE1)**(1.0_sp/(2*km+1))
        end if
        if (k /= 1 .and. (k >= kopt-1 .or. first)) then
            if (errmax < 1.0) exit main_loop
            if (k == kmax .or. k == kopt+1) then
                red=SAFE2/err(km)
                exit
            else if (k == kopt) then
                if (alf(kopt-1,kopt) < err(km)) then
                    red=1.0_sp/err(km)
                    exit
                end if
            else if (kopt == kmax) then
                if (alf(km,kmax-1) < err(km)) then
                    red=alf(km,kmax-1)*SAFE2/err(km)
                    exit
                end if
            else if (alf(km,kopt) < err(km)) then
                red=alf(km,kopt-1)/err(km)
                exit
            end if
        end if
```

```
    end do
    red=max(min(red,REDMIN),REDMAX)
    h=h*red
    reduct=.true.
end do main_loop
x=xnew
hdid=h
first=.false.
kopt=1+iminloc(a(2:km+1)*max(err(1:km),SCALMX))
scale=max(err(kopt-1),SCALMX)
wrkmin=scale*a(kopt)
hnext=h/scale
if (kopt >= k .and. kopt /= kmax .and. .not. reduct) then
    fact=max(scale/alf(kopt-1,kopt),SCALMX)
    if (a(kopt+1)*fact <= wrkmin) then
        hnext=h/fact
        kopt=kopt+1
    end if
end if
END SUBROUTINE stifbs
```

This routine is very similar to `bsstep`, and the same remarks about Fortran 90 constructions on p. 1305 apply here.

Chapter B17. Two Point Boundary Value Problems

```
!   FUNCTION shoot(v) is named "funcv" for use with "newt"
    FUNCTION funcv(v)
    USE nrtype
    USE nr, ONLY : odeint,rkqs
    USE sphoot_caller, ONLY : nvar,x1,x2; USE ode_path, ONLY : xp,yp
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: v
    REAL(SP), DIMENSION(size(v)) :: funcv
    REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Routine for use with `newt` to solve a two point boundary value problem for N coupled ODEs by shooting from `x1` to `x2`. Initial values for the ODEs at `x1` are generated from the n_2 input coefficients `v`, using the user-supplied routine `load`. The routine integrates the ODEs to `x2` using the Runge-Kutta method with tolerance `EPS`, initial stepsize `h1`, and minimum stepsize `hmin`. At `x2` it calls the user-supplied subroutine `score` to evaluate the n_2 functions `funcv` that ought to be zero to satisfy the boundary conditions at `x2`. The functions `funcv` are returned on output. `newt` uses a globally convergent Newton's method to adjust the values of `v` until the functions `funcv` are zero. The user-supplied subroutine `derivs(x,y,dydx)` supplies derivative information to the ODE integrator (see Chapter 16). The module `sphoot_caller` receives its values from the main program so that `funcv` can have the syntax required by `newt`. Set `nvar` $= N$ in the main program.

```
    REAL(SP) :: h1,hmin
    REAL(SP), DIMENSION(nvar) :: y
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs

        SUBROUTINE load(x1,v,y)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x1
        REAL(SP), DIMENSION(:), INTENT(IN) :: v
        REAL(SP), DIMENSION(:), INTENT(OUT) :: y
        END SUBROUTINE load

        SUBROUTINE score(x2,y,f)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x2
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: f
        END SUBROUTINE score
    END INTERFACE
    h1=(x2-x1)/100.0_sp
```

```
    hmin=0.0
    call load(x1,v,y)
    if (associated(xp)) deallocate(xp,yp)          Prevent memory leak if save_steps set
    call odeint(y,x1,x2,EPS,h1,hmin,derivs,rkqs)      to .true.
    call score(x2,y,funcv)
    END FUNCTION funcv
```

⋆ ⋆ ⋆

```
!   FUNCTION shootf(v) is named "funcv" for use with "newt"
    FUNCTION funcv(v)
    USE nrtype
    USE nr, ONLY : odeint,rkqs
    USE sphfpt_caller, ONLY : x1,x2,xf,nn2; USE ode_path, ONLY : xp,yp
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: v
    REAL(SP), DIMENSION(size(v)) :: funcv
    REAL(SP), PARAMETER :: EPS=1.0e-6_sp
```

Routine for use with `newt` to solve a two point boundary value problem for N coupled ODEs by shooting from `x1` and `x2` to a fitting point `xf`. Initial values for the ODEs at `x1` (`x2`) are generated from the n_2 (n_1) coefficients V_1 (V_2), using the user-supplied routine `load1` (`load2`). The coefficients V_1 and V_2 should be stored in a single array `v` of length N in the main program, and referenced by pointers as `v1=>v(1:`n_2`)`, `v2=>v(`n_2+1`:N)`. Here $N = n_1 + n_2$. The routine integrates the ODEs to `xf` using the Runge-Kutta method with tolerance EPS, initial stepsize `h1`, and minimum stepsize `hmin`. At `xf` it calls the user-supplied subroutine `score` to evaluate the N functions `f1` and `f2` that ought to match at `xf`. The differences `funcv` are returned on output. `newt` uses a globally convergent Newton's method to adjust the values of `v` until the functions `funcv` are zero. The user-supplied subroutine `derivs(x,y,dydx)` supplies derivative information to the ODE integrator (see Chapter 16). The module `sphfpt_caller` receives its values from the main program so that `funcv` can have the syntax required by `newt`. Set `nn2` $= n_2$ in the main program.

```
    REAL(SP) :: h1,hmin
    REAL(SP), DIMENSION(size(v)) :: f1,f2,y
    INTERFACE
        SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs

        SUBROUTINE load1(x1,v1,y)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x1
        REAL(SP), DIMENSION(:), INTENT(IN) :: v1
        REAL(SP), DIMENSION(:), INTENT(OUT) :: y
        END SUBROUTINE load1

        SUBROUTINE load2(x2,v2,y)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x2
        REAL(SP), DIMENSION(:), INTENT(IN) :: v2
        REAL(SP), DIMENSION(:), INTENT(OUT) :: y
        END SUBROUTINE load2

        SUBROUTINE score(x2,y,f)
        USE nrtype
        IMPLICIT NONE
        REAL(SP), INTENT(IN) :: x2
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
```

```
    REAL(SP), DIMENSION(:), INTENT(OUT) :: f
    END SUBROUTINE score
END INTERFACE
h1=(x2-x1)/100.0_sp
hmin=0.0
call load1(x1,v,y)                          Path from x1 to xf with best trial values V1.
if (associated(xp)) deallocate(xp,yp)       Prevent memory leak if save_steps set
call odeint(y,x1,xf,EPS,h1,hmin,derivs,rkqs)    to .true.
call score(xf,y,f1)
call load2(x2,v(nn2+1:),y)                  Path from x2 to xf with best trial values V2.
call odeint(y,x2,xf,EPS,h1,hmin,derivs,rkqs)
call score(xf,y,f2)
funcv(:)=f1(:)-f2(:)
END FUNCTION funcv
```

⋆ ⋆ ⋆

```
SUBROUTINE solvde(itmax,conv,slowc,scalv,indexv,nb,y)
USE nrtype; USE nrutil, ONLY : assert_eq,imaxloc,nrerror
USE nr, ONLY : difeq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: itmax,nb
REAL(SP), INTENT(IN) :: conv,slowc
REAL(SP), DIMENSION(:), INTENT(IN) :: scalv
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: y
```

Driver routine for solution of two point boundary value problems with N equations by relaxation. `itmax` is the maximum number of iterations. `conv` is the convergence criterion (see text). `slowc` controls the fraction of corrections actually used after each iteration. `scalv`, a vector of length N, contains typical sizes for each dependent variable, used to weight errors. `indexv`, also of length N, lists the column ordering of variables used to construct the matrix `s` of derivatives. (The `nb` boundary conditions at the first mesh point must contain some dependence on the first `nb` variables listed in `indexv`.) There are a total of M mesh points. `y` is the $N \times M$ array that contains the initial guess for all the dependent variables at each mesh point. On each iteration, it is updated by the calculated correction.

```
INTEGER(I4B) :: ic1,ic2,ic3,ic4,it,j,j1,j2,j3,j4,j5,j6,j7,j8,&
    j9,jc1,jcf,jv,k,k1,k2,km,kp,m,ne,nvars
INTEGER(I4B), DIMENSION(size(scalv)) :: kmax
REAL(SP) :: err,fac
REAL(SP), DIMENSION(size(scalv)) :: ermax
REAL(SP), DIMENSION(size(scalv),2*size(scalv)+1) :: s
REAL(SP), DIMENSION(size(scalv),size(scalv)-nb+1,size(y,2)+1) :: c
ne=assert_eq(size(scalv),size(indexv),size(y,1),'solvde: ne')
m=size(y,2)
k1=1                                        Set up row and column markers.
k2=m
nvars=ne*m
j1=1
j2=nb
j3=nb+1
j4=ne
j5=j4+j1
j6=j4+j2
j7=j4+j3
j8=j4+j4
j9=j8+j1
ic1=1
ic2=ne-nb
ic3=ic2+1
ic4=ne
jc1=1
jcf=ic3
do it=1,itmax                               Primary iteration loop.
    k=k1                                    Boundary conditions at first point.
```

```
    call difeq(k,k1,k2,j9,ic3,ic4,indexv,s,y)
    call pinvs(ic3,ic4,j5,j9,jc1,k1,c,s)
    do k=k1+1,k2                                    Finite difference equations at all point
        kp=k-1                                          pairs.
        call difeq(k,k1,k2,j9,ic1,ic4,indexv,s,y)
        call red(ic1,ic4,j1,j2,j3,j4,j9,ic3,jc1,jcf,kp,c,s)
        call pinvs(ic1,ic4,j3,j9,jc1,k,c,s)
    end do
    k=k2+1                                          Final boundary conditions.
    call difeq(k,k1,k2,j9,ic1,ic2,indexv,s,y)
    call red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jc1,jcf,k2,c,s)
    call pinvs(ic1,ic2,j7,j9,jcf,k2+1,c,s)
    call bksub(ne,nb,jcf,k1,k2,c)                   Backsubstitution.
    do j=1,ne                                       Convergence check, accumulate average
        jv=indexv(j)                                    error.
        km=imaxloc(abs(c(jv,1,k1:k2)))+k1-1
          Find point with largest error, for each dependent variable.
        ermax(j)=c(jv,1,km)
        kmax(j)=km
    end do
    ermax(:)=ermax(:)/scalv(:)                      Weighting for each dependent variable.
    err=sum(sum(abs(c(indexv(:),1,k1:k2)),dim=2)/scalv(:))/nvars
    fac=slowc/max(slowc,err)
      Reduce correction applied when error is large.
    y(:,k1:k2)=y(:,k1:k2)-fac*c(indexv(:),1,k1:k2)      Apply corrections.
    write(*,'(1x,i4,2f12.6)') it,err,fac
      Summary of corrections for this step. Point with largest error for each variable can be
      monitored by writing out kmax and ermax.
    if (err < conv) RETURN
end do
call nrerror('itmax exceeded in solvde')        Convergence failed.
CONTAINS

SUBROUTINE bksub(ne,nb,jf,k1,k2,c)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ne,nb,jf,k1,k2
REAL(SP), DIMENSION(:,:,:), INTENT(INOUT) :: c
   Backsubstitution, used internally by solvde.
INTEGER(I4B) :: im,k,nbf
nbf=ne-nb
im=1
do k=k2,k1,-1
      Use recurrence relations to eliminate remaining dependences.
    if (k == k1) im=nbf+1                       Special handling of first point.
    c(im:ne,jf,k)=c(im:ne,jf,k)-matmul(c(im:ne,1:nbf,k),c(1:nbf,jf,k+1))
end do
c(1:nb,1,k1:k2)=c(1+nbf:nb+nbf,jf,k1:k2)        Reorder corrections to be in column 1.
c(1+nb:nbf+nb,1,k1:k2)=c(1:nbf,jf,k1+1:k2+1)
END SUBROUTINE bksub

SUBROUTINE pinvs(ie1,ie2,je1,jsf,jc1,k,c,s)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ie1,ie2,je1,jsf,jc1,k
REAL(SP), DIMENSION(:,:,:), INTENT(OUT) :: c
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: s
   Diagonalize the square subsection of the s matrix, and store the recursion coefficients in
   c; used internally by solvde.
INTEGER(I4B) :: i,icoff,id,ipiv,jcoff,je2,jp,jpiv,js1
INTEGER(I4B), DIMENSION(ie2) :: indxr
REAL(SP) :: big,piv,pivinv
REAL(SP), DIMENSION(ie2) :: pscl
je2=je1+ie2-ie1
js1=je2+1
pscl(ie1:ie2)=maxval(abs(s(ie1:ie2,je1:je2)),dim=2)
  Implicit pivoting, as in §2.1.
```

```
if (any(pscl(ie1:ie2) == 0.0)) &
    call nrerror('singular matrix, row all 0 in pinvs')
pscl(ie1:ie2)=1.0_sp/pscl(ie1:ie2)
indxr(ie1:ie2)=0
do id=ie1,ie2
    piv=0.0
    do i=ie1,ie2                                  Find pivot element.
        if (indxr(i) == 0) then
            jp=imaxloc(abs(s(i,je1:je2)))+je1-1
            big=abs(s(i,jp))
            if (big*pscl(i) > piv) then
                ipiv=i
                jpiv=jp
                piv=big*pscl(i)
            end if
        end if
    end do
    if (s(ipiv,jpiv) == 0.0) call nrerror('singular matrix in pinvs')
    indxr(ipiv)=jpiv                              In place reduction. Save column order-
    pivinv=1.0_sp/s(ipiv,jpiv)                        ing.
    s(ipiv,je1:jsf)=s(ipiv,je1:jsf)*pivinv        Normalize pivot row.
    s(ipiv,jpiv)=1.0
    do i=ie1,ie2                                  Reduce nonpivot elements in column.
        if (indxr(i) /= jpiv .and. s(i,jpiv) /= 0.0) then
            s(i,je1:jsf)=s(i,je1:jsf)-s(i,jpiv)*s(ipiv,je1:jsf)
            s(i,jpiv)=0.0
        end if
    end do
end do
jcoff=jc1-js1                                     Sort and store unreduced coefficients.
icoff=ie1-je1
c(indxr(ie1:ie2)+icoff,js1+jcoff:jsf+jcoff,k)=s(ie1:ie2,js1:jsf)
END SUBROUTINE pinvs

SUBROUTINE red(iz1,iz2,jz1,jz2,jm1,jm2,jmf,ic1,jc1,jcf,kc,c,s)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iz1,iz2,jz1,jz2,jm1,jm2,jmf,ic1,jc1,jcf,kc
REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: s
REAL(SP), DIMENSION(:,:,:), INTENT(IN) :: c
    Reduce columns jz1-jz2 of the s matrix, using previous results as stored in the c matrix.
    Only columns jm1-jm2,jmf are affected by the prior results. red is used internally by
    solvde.
INTEGER(I4B) :: ic,l,loff
loff=jc1-jm1
ic=ic1
do j=jz1,jz2                                      Loop over columns to be zeroed.
    do l=jm1,jm2                                  Loop over columns altered.
        s(iz1:iz2,l)=s(iz1:iz2,l)-s(iz1:iz2,j)*c(ic,l+loff,kc)
          Loop over rows.
    end do
    s(iz1:iz2,jmf)=s(iz1:iz2,jmf)-s(iz1:iz2,j)*c(ic,jcf,kc)     Plus final element.
    ic=ic+1
end do
END SUBROUTINE red
END SUBROUTINE solvde
```

f90 km=imaxloc... See discussion of imaxloc on p. 1017.

⋆ ⋆ ⋆

```
MODULE sfroid_data                          Communicates with difeq.
USE nrtype
INTEGER(I4B), PARAMETER :: M=41
INTEGER(I4B) :: mm,n
REAL(SP) :: anorm,c2,h
REAL(SP), DIMENSION(M) :: x
END MODULE sfroid_data
```

```
PROGRAM sfroid
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : plgndr,solvde
USE sfroid_data
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NE=3,NB=1
    Sample program using solvde. Computes eigenvalues of spheroidal harmonics S_mn(x;c)
    for m ≥ 0 and n ≥ m. In the program, m is mm, c^2 is c2, and γ of equation (17.4.20)
    is anorm.
INTEGER(I4B) :: itmax
INTEGER(I4B), DIMENSION(NE) :: indexv
REAL(SP) :: conv,slowc
REAL(SP), DIMENSION(M) :: deriv,fac1,fac2
REAL(SP), DIMENSION(NE) :: scalv
REAL(SP), DIMENSION(NE,M) :: y
itmax=100
conv=5.0e-6_sp
slowc=1.0
h=1.0_sp/(M-1)
c2=0.0
write(*,*) 'ENTER M,N'
read(*,*) mm,n
indexv(1:3)=merge( (/ 1, 2, 3 /), (/ 2, 1, 3 /), (mod(n+mm,2) == 1) )
  No interchanges necessary if n+mm is odd; otherwise interchange y_1 and y_2.
anorm=1.0                                   Compute γ.
if (mm /= 0) then
    anorm=(-0.5_sp)**mm*product(&
        arth(n+1,1,mm)*arth(real(n,sp),-1.0_sp,mm)/arth(1,1,mm))
end if
x(1:M-1)=arth(0,1,M-1)*h
fac1(1:M-1)=1.0_sp-x(1:M-1)**2              Compute initial guess.
fac2(1:M-1)=fac1(1:M-1)**(-mm/2.0_sp)
y(1,1:M-1)=plgndr(n,mm,x(1:M-1))*fac2(1:M-1)      P_n^m from §6.8.
deriv(1:M-1)=-((n-mm+1)*plgndr(n+1,mm,x(1:M-1))-(n+1)*&
    x(1:M-1)*plgndr(n,mm,x(1:M-1)))/fac1(1:M-1)
      Derivative of P_n^m from a recurrence relation.
y(2,1:M-1)=mm*x(1:M-1)*y(1,1:M-1)/fac1(1:M-1)+deriv(1:M-1)*fac2(1:M-1)
y(3,1:M-1)=n*(n+1)-mm*(mm+1)
x(M)=1.0                                    Initial guess at x = 1 done separately.
y(1,M)=anorm
y(3,M)=n*(n+1)-mm*(mm+1)
y(2,M)=(y(3,M)-c2)*y(1,M)/(2.0_sp*(mm+1.0_sp))
scalv(1:3)=(/ abs(anorm), max(abs(anorm),y(2,M)), max(1.0_sp,y(3,M)) /)
do
    write (*,*) 'ENTER C**2 OR 999 TO END'
    read (*,*) c2
    if (c2 == 999.0) exit
    call solvde(itmax,conv,slowc,scalv,indexv,NB,y)
    write (*,*) ' M = ',mm,' N = ',n,&
        ' C**2 = ',c2,' LAMBDA = ',y(3,1)+mm*(mm+1)
end do                                      Go back for another value of c^2.
END PROGRAM sfroid
```

`MODULE sfroid_data` This module functions just like a common block to communicate variables with `difeq`. The advantage of a module is that it allows complete specification of the variables.

`anorm=(-0.5_sp)**mm*product(...` This statement computes equation (17.4.20) by direct multiplication.

⋆ ⋆ ⋆

```
SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,s,y)
USE nrtype
USE sfroid_data
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: is1,isf,jsf,k,k1,k2
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: s
REAL(SP), DIMENSION(:,:), INTENT(IN) :: y
   Returns matrix s(i,j) for solvde.
REAL(SP) :: temp,temp2
INTEGER(I4B), DIMENSION(3) :: indexv3
indexv3(1:3)=3+indexv(1:3)
if (k == k1) then            Boundary condition at first point.
    if (mod(n+mm,2) == 1) then
        s(3,indexv3(1:3))= (/ 1.0_sp, 0.0_sp, 0.0_sp /)        Equation (17.4.32).
        s(3,jsf)=y(1,1)                                        Equation (17.4.31).
    else
        s(3,indexv3(1:3))= (/ 0.0_sp, 1.0_sp, 0.0_sp /)        Equation (17.4.32).
        s(3,jsf)=y(2,1)                                        Equation (17.4.31).
    end if
else if (k > k2) then        Boundary conditions at last point.
    s(1,indexv3(1:3))= (/ -(y(3,M)-c2)/(2.0_sp*(mm+1.0_sp)),&
        1.0_sp, -y(1,M)/(2.0_sp*(mm+1.0_sp)) /)                Equation (17.4.35).
    s(1,jsf)=y(2,M)-(y(3,M)-c2)*y(1,M)/(2.0_sp*(mm+1.0_sp))    Equation (17.4.33).
    s(2,indexv3(1:3))=(/ 1.0_sp, 0.0_sp, 0.0_sp /)             Equation (17.4.36).
    s(2,jsf)=y(1,M)-anorm                                      Equation (17.4.34).
else                         Interior point.
    s(1,indexv(1:3))=(/ -1.0_sp, -0.5_sp*h, 0.0_sp /)          Equation (17.4.28).
    s(1,indexv3(1:3))=(/ 1.0_sp, -0.5_sp*h, 0.0_sp /)
    temp=h/(1.0_sp-(x(k)+x(k-1))**2*0.25_sp)
    temp2=0.5_sp*(y(3,k)+y(3,k-1))-c2*0.25_sp*(x(k)+x(k-1))**2
    s(2,indexv(1:3))=(/ temp*temp2*0.5_sp,&                    Equation (17.4.29).
        -1.0_sp-0.5_sp*temp*(mm+1.0_sp)*(x(k)+x(k-1)),&
        0.25_sp*temp*(y(1,k)+y(1,k-1)) /)
    s(2,indexv3(1:3))=s(2,indexv(1:3))
    s(2,indexv3(2))=s(2,indexv3(2))+2.0_sp
    s(3,indexv(1:3))=(/ 0.0_sp, 0.0_sp, -1.0_sp /)             Equation (17.4.30).
    s(3,indexv3(1:3))=(/ 0.0_sp, 0.0_sp, 1.0_sp /)
    s(1,jsf)=y(1,k)-y(1,k-1)-0.5_sp*h*(y(2,k)+y(2,k-1))        Equation (17.4.23).
    s(2,jsf)=y(2,k)-y(2,k-1)-temp*((x(k)+x(k-1))*&             Equation (17.4.24).
        0.5_sp*(mm+1.0_sp)*(y(2,k)+y(2,k-1))-temp2*&
        0.5_sp*(y(1,k)+y(1,k-1)))
    s(3,jsf)=y(3,k)-y(3,k-1)                                   Equation (17.4.27).
end if
END SUBROUTINE difeq
```

⋆ ⋆ ⋆

```
MODULE sphoot_data                          Communicates with load, score, and derivs.
USE nrtype
INTEGER(I4B) :: m,n
REAL(SP) :: c2,dx,gamma
END MODULE sphoot_data
```

```
MODULE sphoot_caller                        Communicates with shoot.
USE nrtype
INTEGER(I4B) :: nvar
REAL(SP) :: x1,x2
END MODULE sphoot_caller
```

```
PROGRAM sphoot
```
Sample program using shoot. Computes eigenvalues of spheroidal harmonics $S_{mn}(x;c)$ for $m \geq 0$ and $n \geq m$. Be sure that routine funcv for newt is provided by shoot (§17.1).
```
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : newt
USE sphoot_data
USE sphoot_caller
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NV=3,N2=1
REAL(SP), DIMENSION(N2) :: v
LOGICAL(LGT) :: check
nvar=NV                                     Number of equations.
dx=1.0e-4_sp                                Avoid evaluating derivatives exactly at x =
do                                              −1.
    write(*,*) 'input m,n,c-squared (999 to end)'
    read(*,*) m,n,c2
    if (c2 == 999.0) exit
    if ((n < m) .or. (m < 0)) cycle
    gamma=(-0.5_sp)**m*product(&            Compute γ of equation (17.4.20).
        arth(n+1,1,m)*(arth(real(n,sp),-1.0_sp,m)/arth(1,1,m)))
    v(1)=n*(n+1)-m*(m+1)+c2/2.0_sp          Initial guess for eigenvalue.
    x1=-1.0_sp+dx                           Set range of integration.
    x2=0.0
    call newt(v,check)                      Find v that zeros function f in score.
    if (check) then
        write(*,*)'shoot failed; bad initial guess'
        exit
    else
        write(*,'(1x,t6,a)') 'mu(m,n)'
        write(*,'(1x,f12.6)') v(1)
    end if
end do
END PROGRAM sphoot
```

```
SUBROUTINE load(x1,v,y)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:), INTENT(IN) :: v
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
```
Supplies starting values for integration at $x = -1 + dx$.
```
REAL(SP) :: y1
y(3)=v(1)
y1=merge(gamma,-gamma, mod(n-m,2) == 0 )
y(2)=-(y(3)-c2)*y1/(2*(m+1))
y(1)=y1+y(2)*dx
END SUBROUTINE load
```

```
SUBROUTINE score(x2,y,f)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: f
    Tests whether boundary condition at x = 0 is satisfied.
f(1)=merge(y(2),y(1), mod(n-m,2) == 0 )
END SUBROUTINE score
```

MODULE sphoot_data...MODULE sphoot_caller These modules function just like common blocks to communicate variables from sphoot to the various subsidiary routines. The advantage of a module is that it allows complete specification of the variables.

```
SUBROUTINE derivs(x,y,dydx)
USE nrtype
USE sphoot_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    Evaluates derivatives for odeint.
dydx(1)=y(2)
dydx(2)=(2.0_sp*x*(m+1.0_sp)*y(2)-(y(3)-c2*x*x)*y(1))/(1.0_sp-x*x)
dydx(3)=0.0
END SUBROUTINE derivs
```

⋆ ⋆ ⋆

```
MODULE sphfpt_data
USE nrtype
INTEGER(I4B) :: m,n
REAL(SP) :: c2,dx,gamma
END MODULE sphfpt_data
```

Communicates with load1, load2, score, and derivs.

```
MODULE sphfpt_caller
USE nrtype
INTEGER(I4B) :: nn2
REAL(SP) :: x1,x2,xf
END MODULE sphfpt_caller
```

Communicates with shootf.

```
PROGRAM sphfpt
```
Sample program using `shootf`. Computes eigenvalues of spheroidal harmonics $S_{mn}(x;c)$ for $m \geq 0$ and $n \geq m$. Be sure that routine `funcv` for `newt` is provided by `shootf` (§17.2). The routine `derivs` is the same as for `sphoot`.
```
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : newt
USE sphfpt_data
USE sphfpt_caller
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: N1=2,N2=1,NTOT=N1+N2
REAL(SP), PARAMETER :: DXX=1.0e-4_sp
REAL(SP), DIMENSION(:), POINTER :: v1,v2
REAL(SP), DIMENSION(NTOT), TARGET :: v
LOGICAL(LGT) :: check
v1=>v(1:N2)
v2=>v(N2+1:NTOT)
nn2=N2
dx=DXX                                  Avoid evaluating derivatives exactly at x =
do                                          ±1.
    write(*,*) 'input m,n,c-squared (999 to end)'
    read(*,*) m,n,c2
    if (c2 == 999.0) exit
    if ((n < m) .or. (m < 0)) cycle
    gamma=(-0.5_sp)**m*product(&        Compute γ of equation (17.4.20).
        arth(n+1,1,m)*(arth(real(n,sp),-1.0_sp,m)/arth(1,1,m)))
    v1(1)=n*(n+1)-m*(m+1)+c2/2.0_sp     Initial guess for eigenvalue and function value.
    v2(2)=v1(1)
    v2(1)=gamma*(1.0_sp-(v2(2)-c2)*dx/(2*(m+1)))
    x1=-1.0_sp+dx                       Set range of integration.
    x2=1.0_sp-dx
    xf=0.0                              Fitting point.
    call newt(v,check)                  Find v that zeros function f in score.
    if (check) then
        write(*,*) 'shootf failed; bad initial guess'
        exit
    else
        write(*,'(1x,t6,a)') 'mu(m,n)'
        write(*,'(1x,f12.6)') v1(1)
    end if
end do
END PROGRAM sphfpt
```

```
SUBROUTINE load1(x1,v1,y)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x1
REAL(SP), DIMENSION(:), INTENT(IN) :: v1
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
```
Supplies starting values for integration at $x = -1 + dx$.
```
REAL(SP) :: y1
y(3)=v1(1)
y1=merge(gamma,-gamma,mod(n-m,2) == 0)
y(2)=-(y(3)-c2)*y1/(2*(m+1))
y(1)=y1+y(2)*dx
END SUBROUTINE load1
```

```
SUBROUTINE load2(x2,v2,y)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x2
REAL(SP), DIMENSION(:), INTENT(IN) :: v2
REAL(SP), DIMENSION(:), INTENT(OUT) :: y
    Supplies starting values for integration at x = 1 − dx.
y(3)=v2(2)
y(1)=v2(1)
y(2)=(y(3)-c2)*y(1)/(2*(m+1))
END SUBROUTINE load2
```

```
SUBROUTINE score(xf,y,f)
USE nrtype
USE sphfpt_data
IMPLICIT NONE
REAL(SP), INTENT(IN) :: xf
REAL(SP), DIMENSION(:), INTENT(IN) :: y
REAL(SP), DIMENSION(:), INTENT(OUT) :: f
    Tests whether solutions match at fitting point x = 0.
f(1:3)=y(1:3)
END SUBROUTINE score
```

f90 `MODULE sphfpt_data...MODULE sphfpt_caller` These modules function just like common blocks to communicate variables from `sphfpt` to the various subsidiary routines. The advantage of a module is that it allows complete specification of the variables.

Chapter B18. Integral Equations and Inverse Theory

```
SUBROUTINE fred2(a,b,t,f,w,g,ak)
USE nrtype; USE nrutil, ONLY : assert_eq,unit_matrix
USE nr, ONLY : gauleg,lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(OUT) :: t,f,w
INTERFACE
    FUNCTION g(t)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t
    REAL(SP), DIMENSION(size(t)) :: g
    END FUNCTION g

    FUNCTION ak(t,s)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
    REAL(SP), DIMENSION(size(t),size(s)) :: ak
    END FUNCTION ak
END INTERFACE
```

Solves a linear Fredholm equation of the second kind by N-point Gaussian quadrature. On input, a and b are the limits of integration. g and ak are user-supplied external functions. g returns $g(t)$ as a vector of length N for a vector of N arguments, while ak returns $\lambda K(t,s)$ as an $N \times N$ matrix. The routine returns arrays t and f of length N containing the abscissas t_i of the Gaussian quadrature and the solution f at these abscissas. Also returned is the array w of length N of Gaussian weights for use with the Nystrom interpolation routine fredin.

```
INTEGER(I4B) :: n
INTEGER(I4B), DIMENSION(size(f)) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(size(f),size(f)) :: omk
n=assert_eq(size(f),size(t),size(w),'fred2')
call gauleg(a,b,t,w)                          Replace gauleg with another routine if not
call unit_matrix(omk)                             using Gauss-Legendre quadrature.
omk=omk-ak(t,t)*spread(w,dim=1,ncopies=n)     Form 1 − λK̃.
f=g(t)
call ludcmp(omk,indx,d)                       Solve linear equations.
call lubksb(omk,indx,f)
END SUBROUTINE fred2
```

f90 `call unit_matrix(omk)` The `unit_matrix` routine in `nrutil` does exactly what its name suggests.

`omk=omk-ak(t,t)*spread(w,dim=1,ncopies=n)` By now this idiom should be second nature: the first column of `ak` gets multiplied by the first element of `w`, and so on.

⋆ ⋆ ⋆

```
FUNCTION fredin(x,a,b,t,f,w,g,ak)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: x,t,f,w
REAL(SP), DIMENSION(size(x)) :: fredin
INTERFACE
    FUNCTION g(t)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t
    REAL(SP), DIMENSION(size(t)) :: g
    END FUNCTION g

    FUNCTION ak(t,s)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
    REAL(SP), DIMENSION(size(t),size(s)) :: ak
    END FUNCTION ak
END INTERFACE
```

Input are arrays `t` and `w` of length N containing the abscissas and weights of the N-point Gaussian quadrature, and the solution array `f` of length N from `fred2`. The function `fredin` returns the array of values of f at an array of points `x` using the Nystrom interpolation formula. On input, `a` and `b` are the limits of integration. `g` and `ak` are user-supplied external functions. `g` returns $g(t)$ as a vector of length N for a vector of N arguments, while `ak` returns $\lambda K(t,s)$ as an $N \times N$ matrix.

```
INTEGER(I4B) :: n
n=assert_eq(size(f),size(t),size(w),'fredin')
fredin=g(x)+matmul(ak(x,t),w*f)
END FUNCTION fredin
```

`fredin=g(x)+matmul...` Fortran 90 allows very concise coding here, which also happens to be much closer to the mathematical formulation than the loops required in Fortran 77.

⋆ ⋆ ⋆

```
SUBROUTINE voltra(t0,h,t,f,g,ak)
USE nrtype; USE nrutil, ONLY : array_copy,assert_eq,unit_matrix
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t0,h
REAL(SP), DIMENSION(:), INTENT(OUT) :: t
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: f
INTERFACE
    FUNCTION g(t)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: t
    REAL(SP), DIMENSION(:), POINTER :: g
    END FUNCTION g

    FUNCTION ak(t,s)
```

```
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: t,s
    REAL(SP), DIMENSION(:,:), POINTER :: ak
    END FUNCTION ak
END INTERFACE
```

Solves a set of M linear Volterra equations of the second kind using the extended trapezoidal rule. On input, t0 is the starting point of the integration. The routine takes $N-1$ steps of size h and returns the abscissas in t, a vector of length N. The solution at these points is returned in the $M \times N$ matrix f. g is a user-supplied external function that returns a pointer to the M-dimensional vector of functions $g_k(t)$, while ak is another user-supplied external function that returns a pointer to the $M \times M$ matrix $K(t,s)$.

```
INTEGER(I4B) :: i,j,n,ncop,nerr,m
INTEGER(I4B), DIMENSION(size(f,1)) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(size(f,1)) :: b
REAL(SP), DIMENSION(size(f,1),size(f,1)) :: a
n=assert_eq(size(f,2),size(t),'voltra: n')
t(1)=t0                                                  Initialize.
call array_copy(g(t(1)),f(:,1),ncop,nerr)
m=assert_eq(size(f,1),ncop,ncop+nerr,'voltra: m')
do i=2,n                                                 Take a step h.
    t(i)=t(i-1)+h
    b=g(t(i))+0.5_sp*h*matmul(ak(t(i),t(1)),f(:,1))      Accumulate right-hand side
    do j=2,i-1                                               of linear equations in b.
        b=b+h*matmul(ak(t(i),t(j)),f(:,j))
    end do
    call unit_matrix(a)                                  Left-hand side goes in ma-
    a=a-0.5_sp*h*ak(t(i),t(i))                               trix a.
    call ludcmp(a,indx,d)                                Solve linear equations.
    call lubksb(a,indx,b)
    f(:,i)=b(:)
end do
END SUBROUTINE voltra
```

f90 `FUNCTION g(t)...REAL(SP), DIMENSION(:), POINTER :: g` The routine voltra requires an argument that is a function returning a vector, but we don't know the dimension of the vector at compile time. The solution is to make the function return a *pointer* to the vector. This is not the same thing as a pointer to a function, which is not allowed in Fortran 90. When you use the pointer in the routine, Fortran 90 figures out from the context that you want the vector of values, so the code remains highly readable. Similarly, the argument ak is a function returning a pointer to a matrix.

The coding of the user-supplied functions g and ak deserves some comment: functions returning pointers to arrays are potential memory leaks if the arrays are allocated dynamically in the functions. Here the user knows in advance the dimension of the problem, and so there is no need to use dynamical allocation in the functions. For example, in a two-dimensional problem, you can code g as follows:

```
FUNCTION g(t)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t
REAL(SP), DIMENSION(:), POINTER :: g
REAL(SP), DIMENSION(2), TARGET, SAVE :: gg
g=>gg
g(1)=...
g(2)=...
END FUNCTION g
```

and similarly for ak.

Suppose, however, we coded g with dynamical allocation:

```
FUNCTION g(t)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t
REAL(SP), DIMENSION(:), POINTER :: g
allocate(g(2))
g(1)=...
g(2)=...
END FUNCTION g
```

Now g never gets deallocated; each time we call the function fresh memory gets consumed. If you have a problem that really does require dynamical allocation in a pointer function, you have to be sure to deallocate the pointer in the calling routine. In voltra, for example, we would declare pointers gtemp and aktemp. Then instead of writing simply

```
b=g(t(i))+...
```

we would write

```
gtemp=>g(t(i))
b=gtemp+...
deallocate(gtemp)
```

and similarly for each pointer function invocation.

`call array_copy(g(t(1)),f(:,1),ncop,nerr)` The routine would work if we replaced this statement with simply `f(:,1)=g(t(1))`. The purpose of using array_copy from nrutil is that we can check that f and g have consistent dimensions with a call to assert_eq.

⋆ ⋆ ⋆

```
FUNCTION wwghts(n,h,kermom)
USE nrtype; USE nrutil, ONLY : geop
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: h
REAL(SP), DIMENSION(n) :: wwghts
INTERFACE
    FUNCTION kermom(y,m)
    USE nrtype
    IMPLICIT NONE
    REAL(DP), INTENT(IN) :: y
    INTEGER(I4B), INTENT(IN) :: m
    REAL(DP), DIMENSION(m) :: kermom
    END FUNCTION kermom
END INTERFACE
```

Returns in wwghts(1:n) weights for the n-point equal-interval quadrature from 0 to $(n-1)h$ of a function $f(x)$ times an arbitrary (possibly singular) weight function $w(x)$ whose indefinite-integral moments $F_n(y)$ are provided by the user-supplied function kermom.

```
INTEGER(I4B) :: j
REAL(DP) :: hh,hi,c,a,b
REAL(DP), DIMENSION(4) :: wold,wnew,w
hh=h                                  Double precision on internal calculations even though
hi=1.0_dp/hh                              the interface is in single precision.
wwghts(1:n)=0.0                       Zero all the weights so we can sum into them.
wold(1:4)=kermom(0.0_dp,4)            Evaluate indefinite integrals at lower end.
```

```
if (n >= 4) then                          Use highest available order.
    b=0.0                                 For another problem, you might change this lower
    do j=1,n-3                                limit.
        c=j-1                             This is called k in equation (18.3.5).
        a=b                               Set upper and lower limits for this step.
        b=a+hh
        if (j == n-3) b=(n-1)*hh          Last interval: go all the way to end.
        wnew(1:4)=kermom(b,4)
        w(1:4)=(wnew(1:4)-wold(1:4))*geop(1.0_dp,hi,4)      Equation (18.3.4).
        wwghts(j:j+3)=wwghts(j:j+3)+(/&                     Equation (18.3.5).
            ((c+1.0_dp)*(c+2.0_dp)*(c+3.0_dp)*w(1)&
            -(11.0_dp+c*(12.0_dp+c*3.0_dp))*w(2)&
                +3.0_dp*(c+2.0_dp)*w(3)-w(4))/6.0_dp,&
            (-c*(c+2.0_dp)*(c+3.0_dp)*w(1)&
            +(6.0_dp+c*(10.0_dp+c*3.0_dp))*w(2)&
                -(3.0_dp*c+5.0_dp)*w(3)+w(4))*0.50_dp,&
            (c*(c+1.0_dp)*(c+3.0_dp)*w(1)&
            -(3.0_dp+c*(8.0_dp+c*3.0_dp))*w(2)&
                +(3.0_dp*c+4.0_dp)*w(3)-w(4))*0.50_dp,&
            (-c*(c+1.0_dp)*(c+2.0_dp)*w(1)&
            +(2.0_dp+c*(6.0_dp+c*3.0_dp))*w(2)&
            -3.0_dp*(c+1.0_dp)*w(3)+w(4))/6.0_dp /)
        wold(1:4)=wnew(1:4)               Reset lower limits for moments.
    end do
else if (n == 3) then                     Lower-order cases; not recommended.
    wnew(1:3)=kermom(hh+hh,3)
    w(1:3)= (/ wnew(1)-wold(1), hi*(wnew(2)-wold(2)),&
        hi**2*(wnew(3)-wold(3)) /)
    wwghts(1:3)= (/ w(1)-1.50_dp*w(2)+0.50_dp*w(3),&
        2.0_dp*w(2)-w(3), 0.50_dp*(w(3)-w(2)) /)
else if (n == 2) then
    wnew(1:2)=kermom(hh,2)
    wwghts(2)=hi*(wnew(2)-wold(2))
    wwghts(1)=wnew(1)-wold(1)-wwghts(2)
end if
END FUNCTION wwghts
```

⋆ ⋆ ⋆

```
MODULE kermom_info
USE nrtype
REAL(DP) :: kermom_x
END MODULE kermom_info
```

```
FUNCTION kermom(y,m)
USE nrtype
USE kermom_info
IMPLICIT NONE
REAL(DP), INTENT(IN) :: y
INTEGER(I4B), INTENT(IN) :: m
REAL(DP), DIMENSION(m) :: kermom
```

Returns in kermom(1:m) the first m indefinite-integral moments of one row of the singular part of the kernel. (For this example, m is hard-wired to be 4.) The input variable y labels the column, while kermom_x (in the module kermom_info) is the row.

```
REAL(DP) :: x,d,df,clog,x2,x3,x4
x=kermom_x                          We can take x as the lower limit of integration. Thus, we
if (y >= x) then                        return the moment integrals either purely to the left or
    d=y-x                               purely to the right of the diagonal.
    df=2.0_dp*sqrt(d)*d
    kermom(1:4) = (/ df/3.0_dp, df*(x/3.0_dp+d/5.0_dp),&
```

```
        df*((x/3.0_dp + 0.4_dp*d)*x + d**2/7.0_dp),&
        df*(((x/3.0_dp + 0.6_dp*d)*x + 3.0_dp*d**2/7.0_dp)*x&
            + d**3/9.0_dp) /)
else
    x2=x**2
    x3=x2*x
    x4=x2*x2
    d=x-y
    clog=log(d)
    kermom(1:4) = (/ d*(clog-1.0_dp),&
        -0.25_dp*(3.0_dp*x+y-2.0_dp*clog*(x+y))*d,&
        (-11.0_dp*x3+y*(6.0_dp*x2+y*(3.0_dp*x+2.0_dp*y))&
            +6.0_dp*clog*(x3-y**3))/18.0_dp,&
        (-25.0_dp*x4+y*(12.0_dp*x3+y*(6.0_dp*x2+y*&
            (4.0_dp*x+3.0_dp*y)))+12.0_dp*clog*(x4-y**4))/48.0_dp /)
end if
END FUNCTION kermom
```

f90 `MODULE kermom_info` This module functions just like a common block to share the variable `kermom_x` with the routine `quadmx`.

⋆ ⋆ ⋆

```
SUBROUTINE quadmx(a)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,diagadd,outerprod
USE nr, ONLY : wwghts,kermom
USE kermom_info
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: a
```

Constructs in the $N \times N$ array a the quadrature matrix for an example Fredholm equation of the second kind. The nonsingular part of the kernel is computed within this routine, while the quadrature weights that integrate the singular part of the kernel are obtained via calls to `wwghts`. An external routine `kermom`, which supplies indefinite-integral moments of the singular part of the kernel, is passed to `wwghts`.

```
INTEGER(I4B) :: j,n
REAL(SP) :: h,x
REAL(SP), DIMENSION(size(a,1)) :: wt
n=assert_eq(size(a,1),size(a,2),'quadmx')
h=PI/(n-1)
do j=1,n
    x=(j-1)*h
    kermom_x=x                      Put x in the module kermom_info for use by kermom.
    wt(:)=wwghts(n,h,kermom)        Part of nonsingular kernel.
    a(j,:)=wt(:)                    Put together all the pieces of the kernel.
end do
wt(:)=cos(arth(0,1,n)*h)
a(:,:)=a(:,:)*outerprod(wt(:),wt(:))
call diagadd(a,1.0_sp)              Since equation of the second kind, there is diagonal
END SUBROUTINE quadmx                   piece independent of h.
```

f90 `call diagadd...` See discussion of `diagadd` after `hqr` on p. 1234.

⋆ ⋆ ⋆

```
PROGRAM fredex
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : quadmx,ludcmp,lubksb
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: N=40
INTEGER(I4B) :: j
INTEGER(I4B), DIMENSION(N) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(N) :: g,x
REAL(SP), DIMENSION(N,N) :: a
```

This sample program shows how to solve a Fredholm equation of the second kind using the product Nystrom method and a quadrature rule especially constructed for a particular, singular, kernel.

Parameter: N is the size of the grid.

```
call quadmx(a)                Make the quadrature matrix; all the action is here.
call ludcmp(a,indx,d)         Decompose the matrix.
x(:)=arth(0,1,n)*PI/(n-1)
g(:)=sin(x(:))                Construct the right-hand side, here sin x.
call lubksb(a,indx,g)         Backsubstitute.
do j=1,n                      Write out the solution.
    write (*,*) j,x(j),g(j)
end do
write (*,*) 'normal completion'
END PROGRAM fredex
```

Chapter B19. Partial Differential Equations

```
SUBROUTINE sor(a,b,c,d,e,f,u,rjac)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: a,b,c,d,e,f
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), INTENT(IN) :: rjac
INTEGER(I4B), PARAMETER :: MAXITS=1000
REAL(DP), PARAMETER :: EPS=1.0e-5_dp
```

Successive overrelaxation solution of equation (19.5.25) with Chebyshev acceleration. a, b, c, d, e, and f are input as the coefficients of the equation, each dimensioned to the grid size $J \times J$. u is input as the initial guess to the solution, usually zero, and returns with the final value. rjac is input as the spectral radius of the Jacobi iteration, or an estimate of it. Double precision is a good idea for J bigger than about 25.

```
REAL(DP), DIMENSION(size(a,1),size(a,1)) :: resid
INTEGER(I4B) :: jmax,jm1,jm2,jm3,n
REAL(DP) :: anorm,anormf,omega
jmax=assert_eq((/size(a,1),size(a,2),size(b,1),size(b,2), &
    size(c,1),size(c,2),size(d,1),size(d,2),size(e,1), &
    size(e,2),size(f,1),size(f,2),size(u,1),size(u,2)/),'sor')
jm1=jmax-1
jm2=jmax-2
jm3=jmax-3
anormf=sum(abs(f(2:jm1,2:jm1)))
```

Compute initial norm of residual and terminate iteration when norm has been reduced by a factor EPS. This computation assumes initial u is zero.

```
omega=1.0
do n=1,MAXITS
```

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:

```
    resid(2:jm1:2,2:jm1:2)=a(2:jm1:2,2:jm1:2)*u(3:jmax:2,2:jm1:2)+&
        b(2:jm1:2,2:jm1:2)*u(1:jm2:2,2:jm1:2)+&
        c(2:jm1:2,2:jm1:2)*u(2:jm1:2,3:jmax:2)+&
        d(2:jm1:2,2:jm1:2)*u(2:jm1:2,1:jm2:2)+&
        e(2:jm1:2,2:jm1:2)*u(2:jm1:2,2:jm1:2)-f(2:jm1:2,2:jm1:2)
    u(2:jm1:2,2:jm1:2)=u(2:jm1:2,2:jm1:2)-omega*&
        resid(2:jm1:2,2:jm1:2)/e(2:jm1:2,2:jm1:2)
    resid(3:jm2:2,3:jm2:2)=a(3:jm2:2,3:jm2:2)*u(4:jm1:2,3:jm2:2)+&
        b(3:jm2:2,3:jm2:2)*u(2:jm3:2,3:jm2:2)+&
        c(3:jm2:2,3:jm2:2)*u(3:jm2:2,4:jm1:2)+&
        d(3:jm2:2,3:jm2:2)*u(3:jm2:2,2:jm3:2)+&
        e(3:jm2:2,3:jm2:2)*u(3:jm2:2,3:jm2:2)-f(3:jm2:2,3:jm2:2)
    u(3:jm2:2,3:jm2:2)=u(3:jm2:2,3:jm2:2)-omega*&
        resid(3:jm2:2,3:jm2:2)/e(3:jm2:2,3:jm2:2)
    omega=merge(1.0_dp/(1.0_dp-0.5_dp*rjac**2), &
        1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega), n == 1)
```

Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:

```
    resid(3:jm2:2,2:jm1:2)=a(3:jm2:2,2:jm1:2)*u(4:jm1:2,2:jm1:2)+&
        b(3:jm2:2,2:jm1:2)*u(2:jm3:2,2:jm1:2)+&
```

```
        c(3:jm2:2,2:jm1:2)*u(3:jm2:2,3:jmax:2)+&
        d(3:jm2:2,2:jm1:2)*u(3:jm2:2,1:jm2:2)+&
        e(3:jm2:2,2:jm1:2)*u(3:jm2:2,2:jm1:2)-f(3:jm2:2,2:jm1:2)
    u(3:jm2:2,2:jm1:2)=u(3:jm2:2,2:jm1:2)-omega*&
        resid(3:jm2:2,2:jm1:2)/e(3:jm2:2,2:jm1:2)
    resid(2:jm1:2,3:jm2:2)=a(2:jm1:2,3:jm2:2)*u(3:jmax:2,3:jm2:2)+&
        b(2:jm1:2,3:jm2:2)*u(1:jm2:2,3:jm2:2)+&
        c(2:jm1:2,3:jm2:2)*u(2:jm1:2,4:jm1:2)+&
        d(2:jm1:2,3:jm2:2)*u(2:jm1:2,2:jm3:2)+&
        e(2:jm1:2,3:jm2:2)*u(2:jm1:2,3:jm2:2)-f(2:jm1:2,3:jm2:2)
    u(2:jm1:2,3:jm2:2)=u(2:jm1:2,3:jm2:2)-omega*&
        resid(2:jm1:2,3:jm2:2)/e(2:jm1:2,3:jm2:2)
    omega=1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega)
    anorm=sum(abs(resid(2:jm1,2:jm1)))
    if (anorm < EPS*anormf) exit
end do
if (n > MAXITS) call nrerror('MAXITS exceeded in sor')
END SUBROUTINE sor
```

Red-black iterative schemes like the one used in `sor` are easily parallelizable. Updating the red grid points requires information only from the black grid points, so they can all be updated independently. Similarly the black grid points can all be updated independently. Since nearest neighbors are involved in the updating, communication costs can be kept to a minimum.

There are several possibilities for coding the red-black iteration in a data parallel way using only Fortran 90 and no parallel language extensions. One way is to define an $N \times N$ logical mask `red` that is true on the red grid points and false on the black. Then each iteration consists of an update governed by a `where(red)...end where` block and a `where(.not. red)...end where` block. We have chosen a more direct coding that avoids the need for storage of the array `red`. The red update corresponds to the even-even and odd-odd grid points, the black to the even-odd and odd-even points. We can code each of these four cases directly with array sections, as in the routine above.

The array section notation used in `sor` is rather dense and hard to read. We could use pointer aliases to try to simplify things, but since each array section is different, we end up merely giving names to each term that was there all along. Pointer aliases do help if we code `sor` using a logical mask. Since there may be machines on which this version is faster, and since it is of some pedagogic interest, we give the alternative code:

```
SUBROUTINE sor_mask(a,b,c,d,e,f,u,rjac)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), TARGET, INTENT(IN) :: a,b,c,d,e,f
REAL(DP), DIMENSION(:,:), TARGET, INTENT(INOUT) :: u
REAL(DP), INTENT(IN) :: rjac
INTEGER(I4B), PARAMETER :: MAXITS=1000
REAL(DP), PARAMETER :: EPS=1.0e-5_dp
REAL(DP), DIMENSION(:,:), ALLOCATABLE :: resid
REAL(DP), DIMENSION(:,:), POINTER :: u_int,u_down,u_up,u_left,&
    u_right,a_int,b_int,c_int,d_int,e_int,f_int
INTEGER(I4B) :: jmax,jm1,jm2,jm3,n
REAL(DP) anorm,anormf,omega
LOGICAL, DIMENSION(:,:), ALLOCATABLE :: red
jmax=assert_eq((/size(a,1),size(a,2),size(b,1),size(b,2), &
```

```
    size(c,1),size(c,2),size(d,1),size(d,2),size(e,1), &
    size(e,2),size(f,1),size(f,2),size(u,1),size(u,2)/),'sor')
jm1=jmax-1
jm2=jmax-2
jm3=jmax-3
allocate(resid(jm2,jm2),red(jm2,jm2))          Interior is (jmax − 2) × (jmax − 2).
red=.false.
red(1:jm2:2,1:jm2:2)=.true.
red(2:jm3:2,2:jm3:2)=.true.
u_int=>u(2:jm1,2:jm1)
u_down=>u(3:jmax,2:jm1)
u_up=>u(1:jm2,2:jm1)
u_left=>u(2:jm1,1:jm2)
u_right=>u(2:jm1,3:jmax)
a_int=>a(2:jm1,2:jm1)
b_int=>b(2:jm1,2:jm1)
c_int=>c(2:jm1,2:jm1)
d_int=>d(2:jm1,2:jm1)
e_int=>e(2:jm1,2:jm1)
f_int=>f(2:jm1,2:jm1)
anormf=sum(abs(f_int))
omega=1.0
do n=1,MAXITS
    where(red)
        resid=a_int*u_down+b_int*u_up+c_int*u_right+&
            d_int*u_left+e_int*u_int-f_int
        u_int=u_int-omega*resid/e_int
    end where
    omega=merge(1.0_dp/(1.0_dp-0.5_dp*rjac**2), &
        1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega), n == 1)
    where(.not.red)
        resid=a_int*u_down+b_int*u_up+c_int*u_right+&
            d_int*u_left+e_int*u_int-f_int
        u_int=u_int-omega*resid/e_int
    end where
    omega=1.0_dp/(1.0_dp-0.25_dp*rjac**2*omega)
    anorm=sum(abs(resid))
    if(anorm < EPS*anormf)exit
end do
deallocate(resid,red)
if (n > MAXITS) call nrerror('MAXITS exceeded in sor')
END SUBROUTINE sor_mask
```

⋆ ⋆ ⋆

```
SUBROUTINE mglin(u,ncycle)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : interp,rstrct,slvsml
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: ncycle
```

Full Multigrid Algorithm for solution of linear elliptic equation, here the model problem (19.0.6). On input u contains the right-hand side ρ in an $N \times N$ array, while on output it returns the solution. The dimension N is related to the number of grid levels used in the solution, ng below, by $N =$ 2**ng+1. ncycle is the number of V-cycles to be used at each level.

```
INTEGER(I4B) :: j,jcycle,n,ng,ngrid,nn
TYPE ptr2d                                  Define a type so we can have an array of pointers
    REAL(DP), POINTER :: a(:,:)                 to arrays of grid variables.
END TYPE ptr2d
TYPE(ptr2d), ALLOCATABLE :: rho(:)
```

```
REAL(DP), DIMENSION(:,:), POINTER :: uj,uj_1
n=assert_eq(size(u,1),size(u,2),'mglin')
ng=nint(log(n-1.0)/log(2.0))
if (n /= 2**ng+1) call nrerror('n-1 must be a power of 2 in mglin')
allocate(rho(ng))
nn=n
ngrid=ng
allocate(rho(ngrid)%a(nn,nn))          Allocate storage for r.h.s. on grid ng,
rho(ngrid)%a=u                         and fill it with the input r.h.s.
do                                     Similarly allocate storage and fill r.h.s. on all coarse
    if (nn <= 3) exit                      grids by restricting from finer grids.
    nn=nn/2+1
    ngrid=ngrid-1
    allocate(rho(ngrid)%a(nn,nn))
    rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
end do
nn=3
allocate(uj(nn,nn))
call slvsml(uj,rho(1)%a)               Initial solution on coarsest grid.
do j=2,ng                              Nested iteration loop.
    nn=2*nn-1
    uj_1=>uj
    allocate(uj(nn,nn))
    uj=interp(uj_1)                    Interpolate from grid j-1 to next finer grid j.
    deallocate(uj_1)
    do jcycle=1,ncycle                 V-cycle loop.
        call mg(j,uj,rho(j)%a)
    end do
end do
u=uj                                   Return solution in u.
deallocate(uj)
do j=1,ng
    deallocate(rho(j)%a)
end do
deallocate(rho)
CONTAINS

RECURSIVE SUBROUTINE mg(j,u,rhs)
USE nrtype
USE nr, ONLY : interp,relax,resid,rstrct,slvsml
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: j
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
INTEGER(I4B), PARAMETER :: NPRE=1,NPOST=1
    Recursive multigrid iteration. On input, j is the current level, u is the current value of the
    solution, and rhs is the right-hand side. On output u contains the improved solution at the
    current level.
    Parameters: NPRE and NPOST are the number of relaxation sweeps before and after the
    coarse-grid correction is computed.
INTEGER(I4B) :: jpost,jpre
REAL(DP), DIMENSION((size(u,1)+1)/2,(size(u,1)+1)/2) :: res,v
if (j == 1) then                       Bottom of V: Solve on coarsest grid.
    call slvsml(u,rhs)
else                                   On downward stoke of the V.
    do jpre=1,NPRE                     Pre-smoothing.
        call relax(u,rhs)
    end do
    res=rstrct(resid(u,rhs))           Restriction of the residual is the next r.h.s.
    v=0.0                              Zero for initial guess in next relaxation.
    call mg(j-1,v,res)                 Recursive call for the coarse grid correction.
    u=u+interp(v)                      On upward stroke of V.
    do jpost=1,NPOST                   Post-smoothing.
        call relax(u,rhs)
```

```
    end do
end if
END SUBROUTINE mg
END SUBROUTINE mglin
```

The Fortran 90 version of `mglin` (and of `mgfas` below) is quite different from the Fortran 77 version, although the algorithm is identical. First, we use a recursive implementation. This makes the code much more transparent. It also makes the memory management much better: we simply define the new arrays `res` and `v` as automatic arrays of the appropriate dimension on each recursive call to a coarser level. And a third benefit is that it is trivial to change the code to increase the number of multigrid iterations done at level $j-1$ by each iteration at level j, i.e., to set the quantity γ in §19.6 to a value greater than one. (Recall that $\gamma = 1$ as chosen in `mglin` gives V-cycles, $\gamma = 2$ gives W-cycles.) Simply enclose the recursive call in a do-loop:

```
do i=1,merge(gamma,1,j /= 2)
   call mg(j-1,v,res)
end do
```

The merge expression ensures that there is no more than one call to the coarsest level, where the problem is solved exactly.

A second improvement in the Fortran 90 version is to make the procedures `resid`, `interp`, and `rstrct` functions instead of subroutines. This allows us to code the algorithm exactly as written mathematically.

`TYPE ptr2d...` The right-hand-side quantity ρ is supplied initially on the finest grid in the argument `u`. It has to be defined on the coarser grids by restriction, and then supplied as the right-hand side to `mg` in the nested iteration loop. This loop starts at the coarsest level and progresses up to the finest level. We thus need a data structure to store ρ on all the grid levels. A convenient way to implement this in Fortran 90 is to define a type `ptr2d`, a pointer to a two-dimensional array `a` that represents a grid. (In three dimensions, `a` would of course be three-dimensional.) We then declare the variable ρ as an allocatable array of type `ptr2d`:

```
TYPE(ptr2d), ALLOCATABLE :: rho(:)
```

Next we allocate storage for ρ on each level. The number of levels or grids, `ng`, is known only at run time:

```
allocate(rho(ng))
```

Then we allocate storage as needed on particular sized grids. For example,

```
allocate(rho(ngrid)%a(nn,nn))
```

allocates an `nn` $\times$ `nn` grid for `rho` on grid number `ngrid`.

The various subsidiary routines of `mglin` such as `rstrct` and `interp` are written to accept two-dimensional arrays as arguments. With the data structure we've employed, using these routines is simple. For example,

```
rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
```

will restrict `rho` from the grid `ngrid+1` to the grid `ngrid`. The statement is even more readable if we mentally ignore the `%a` that is tagged onto each variable. (If

we actually did omit %a in the code, the compiler would think we meant the array of type ptr2d instead of the grid array.)

Note that while Fortran 90 does not allow you to declare an array of pointers directly, you can achieve the same effect by declaring your own type, as we have done with ptr2d in this example.

```
FUNCTION rstrct(uf)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: uf
REAL(DP), DIMENSION((size(uf,1)+1)/2,(size(uf,1)+1)/2) :: rstrct
```

Half-weighting restriction. If N_c is the coarse-grid dimension, the fine-grid solution is input in the $(2N_c-1)\times(2N_c-1)$ array uf, the coarse-grid solution is returned in the $N_c \times N_c$ array rstrct.

```
INTEGER(I4B) :: nc,nf
nf=assert_eq(size(uf,1),size(uf,2),'rstrct')
nc=(nf+1)/2
rstrct(2:nc-1,2:nc-1)=0.5_dp*uf(3:nf-2:2,3:nf-2:2)+0.125_dp*(&     Interior points.
    uf(4:nf-1:2,3:nf-2:2)+uf(2:nf-3:2,3:nf-2:2)+&
    uf(3:nf-2:2,4:nf-1:2)+uf(3:nf-2:2,2:nf-3:2))
rstrct(1:nc,1)=uf(1:nf:2,1)                                        Boundary points.
rstrct(1:nc,nc)=uf(1:nf:2,nf)
rstrct(1,1:nc)=uf(1,1:nf:2)
rstrct(nc,1:nc)=uf(nf,1:nf:2)
END FUNCTION rstrct
```

```
FUNCTION interp(uc)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: uc
REAL(DP), DIMENSION(2*size(uc,1)-1,2*size(uc,1)-1) :: interp
```

Coarse-to-fine prolongation by bilinear interpolation. If N_f is the fine-grid dimension and N_c the coarse-grid dimension, then $N_f = 2N_c - 1$. The coarse-grid solution is input as uc, the fine-grid solution is returned in interp.

```
INTEGER(I4B) :: nc,nf
nc=assert_eq(size(uc,1),size(uc,2),'interp')
nf=2*nc-1
interp(1:nf:2,1:nf:2)=uc(1:nc,1:nc)
  Do elements that are copies.
interp(2:nf-1:2,1:nf:2)=0.5_dp*(interp(3:nf:2,1:nf:2)+ &
    interp(1:nf-2:2,1:nf:2))
      Do odd-numbered columns, interpolating vertically.
interp(1:nf,2:nf-1:2)=0.5_dp*(interp(1:nf,3:nf:2)+interp(1:nf,1:nf-2:2))
  Do even-numbered columns, interpolating horizontally.
END FUNCTION interp
```

```
SUBROUTINE slvsml(u,rhs)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
```

Solution of the model problem on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in rhs(1:3,1:3) and the solution is returned in u(1:3,1:3).

```
REAL(DP) :: h
u=0.0
h=0.5_dp
u(2,2)=-h*h*rhs(2,2)/4.0_dp
END SUBROUTINE slvsml
```

```
SUBROUTINE relax(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
```

Red-black Gauss-Seidel relaxation for model problem. The current value of the solution `u` is updated, using the right-hand-side function `rhs`. `u` and `rhs` are square arrays of the same odd dimension.

```
INTEGER(I4B) :: n
REAL(DP) :: h,h2
n=assert_eq(size(u,1),size(u,2),size(rhs,1),size(rhs,2),'relax')
h=1.0_dp/(n-1)
h2=h*h
```

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:

```
u(2:n-1:2,2:n-1:2)=0.25_dp*(u(3:n:2,2:n-1:2)+u(1:n-2:2,2:n-1:2)+&
    u(2:n-1:2,3:n:2)+u(2:n-1:2,1:n-2:2)-h2*rhs(2:n-1:2,2:n-1:2))
u(3:n-2:2,3:n-2:2)=0.25_dp*(u(4:n-1:2,3:n-2:2)+u(2:n-3:2,3:n-2:2)+&
    u(3:n-2:2,4:n-1:2)+u(3:n-2:2,2:n-3:2)-h2*rhs(3:n-2:2,3:n-2:2))
```

Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:

```
u(3:n-2:2,2:n-1:2)=0.25_dp*(u(4:n-1:2,2:n-1:2)+u(2:n-3:2,2:n-1:2)+&
    u(3:n-2:2,3:n:2)+u(3:n-2:2,1:n-2:2)-h2*rhs(3:n-2:2,2:n-1:2))
u(2:n-1:2,3:n-2:2)=0.25_dp*(u(3:n:2,3:n-2:2)+u(1:n-2:2,3:n-2:2)+&
    u(2:n-1:2,4:n-1:2)+u(2:n-1:2,2:n-3:2)-h2*rhs(2:n-1:2,3:n-2:2))
END SUBROUTINE relax
```

See the discussion of red-black relaxation after `sor` on p. 1333.

```
FUNCTION resid(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,rhs
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: resid
```

Returns *minus* the residual for the model problem. Input quantities are `u` and `rhs`, while the residual is returned in `resid`. All three quantities are square arrays with the same odd dimension.

```
INTEGER(I4B) :: n
REAL(DP) :: h,h2i
n=assert_eq((/size(u,1),size(u,2),size(rhs,1),size(rhs,2)/),'resid')
n=size(u,1)
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)
resid(2:n-1,2:n-1)=-h2i*(u(3:n,2:n-1)+u(1:n-2,2:n-1)+u(2:n-1,3:n)+&
    u(2:n-1,1:n-2)-4.0_dp*u(2:n-1,2:n-1))+rhs(2:n-1,2:n-1)      Interior points.
resid(1:n,1)=0.0                                                Boundary points.
resid(1:n,n)=0.0
resid(1,1:n)=0.0
resid(n,1:n)=0.0
END FUNCTION resid
```

⋆ ⋆ ⋆

```
SUBROUTINE mgfas(u,maxcyc)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : interp,lop,rstrct,slvsm2
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
INTEGER(I4B), INTENT(IN) :: maxcyc
```

Full Multigrid Algorithm for FAS solution of nonlinear elliptic equation, here equation (19.6.44). On input u contains the right-hand side ρ in an $N \times N$ array, while on output it returns the solution. The dimension N is related to the number of grid levels used in the solution, ng below, by $N = 2^{**}ng+1$. maxcyc is the maximum number of V-cycles to be used at each level.

```
INTEGER(I4B) :: j,jcycle,n,ng,ngrid,nn
REAL(DP) :: res,trerr
TYPE ptr2d                                  Define a type so we can have an array of
    REAL(DP), POINTER :: a(:,:)                 pointers to arrays of grid variables.
END TYPE ptr2d
TYPE(ptr2d), ALLOCATABLE :: rho(:)
REAL(DP), DIMENSION(:,:), POINTER :: uj,uj_1
n=assert_eq(size(u,1),size(u,2),'mgfas')
ng=nint(log(n-1.0)/log(2.0))
if (n /= 2**ng+1) call nrerror('n-1 must be a power of 2 in mgfas')
allocate(rho(ng))
nn=n
ngrid=ng
allocate(rho(ngrid)%a(nn,nn))               Allocate storage for r.h.s. on grid ng,
rho(ngrid)%a=u                              and fill it with ρ from the fine grid.
do                                          Similarly allocate storage and fill r.h.s. by re-
    if (nn <= 3) exit                           striction on all coarse grids.
    nn=nn/2+1
    ngrid=ngrid-1
    allocate(rho(ngrid)%a(nn,nn))
    rho(ngrid)%a=rstrct(rho(ngrid+1)%a)
end do
nn=3
allocate(uj(nn,nn))
call slvsm2(uj,rho(1)%a)                    Initial solution on coarsest grid.
do j=2,ng                                   Nested iteration loop.
    nn=2*nn-1
    uj_1=>uj
    allocate(uj(nn,nn))
    uj=interp(uj_1)                         Interpolate from grid j-1 to next finer grid
    deallocate(uj_1)                            j.
    do jcycle=1,maxcyc                      V-cycle loop.
        call mg(j,uj,trerr=trerr)
        res=sqrt(sum((lop(uj)-rho(j)%a)**2))/nn      Form residual ‖d_h‖.
        if (res < trerr) exit               No more V-cycles needed if residual small
    end do                                      enough.
end do
u=uj                                        Return solution in u.
deallocate(uj)
do j=1,ng
    deallocate(rho(j)%a)
end do
deallocate(rho)
CONTAINS

RECURSIVE SUBROUTINE mg(j,u,rhs,trerr)
USE nrtype
USE nr, ONLY : interp,lop,relax2,rstrct,slvsm2
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: j
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN), OPTIONAL :: rhs
REAL(DP), INTENT(OUT), OPTIONAL :: trerr
```

```
INTEGER(I4B), PARAMETER :: NPRE=1,NPOST=1
REAL(DP), PARAMETER :: ALPHA=0.33_dp
```

Recursive multigrid iteration. On input, `j` is the current level and `u` is the current value of the solution. For the first call on a given level, the right-hand side is zero, and the optional argument `rhs` is not present. Subsequent recursive calls supply a nonzero `rhs` as in equation (19.6.33). On output `u` contains the improved solution at the current level. When the first call on a given level is made, the relative truncation error τ is returned in the optional argument `trerr`.

Parameters: `NPRE` and `NPOST` are the number of relaxation sweeps before and after the coarse-grid correction is computed; `ALPHA` relates the estimated truncation error to the norm of the residual.

```
INTEGER(I4B) :: jpost,jpre
REAL(DP), DIMENSION((size(u,1)+1)/2,(size(u,1)+1)/2) :: v,ut,tau
if (j == 1) then                                  Bottom of V: Solve on coarsest grid.
    call slvsm2(u,rhs+rho(j)%a)
else                                              On downward stoke of the V.
    do jpre=1,NPRE                                Pre-smoothing.
        if (present(rhs)) then
            call relax2(u,rhs+rho(j)%a)
        else
            call relax2(u,rho(j)%a)
        end if
    end do
    ut=rstrct(u)                                  $\mathcal{R}\widetilde{u}_h$.
    v=ut                                          Make a copy in v.
    if (present(rhs)) then
        tau=lop(ut)-rstrct(lop(u)-rhs)            Form $\widetilde{\tau}_h + f_H = \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}\mathcal{L}_h(\widetilde{u}_h) +$
    else                                              $f_H$.
        tau=lop(ut)-rstrct(lop(u))
        trerr=ALPHA*sqrt(sum(tau**2))/size(tau,1)     Estimate truncation error $\tau$.
    end if
    call mg(j-1,v,tau)                            Recursive call for the coarse-grid correction.
    u=u+interp(v-ut)                              $\widetilde{u}_h^{\rm new} = \widetilde{u}_h + \mathcal{P}(\widetilde{u}_H - \mathcal{R}\widetilde{u}_h)$
    do jpost=1,NPOST                              Post-smoothing.
        if (present(rhs)) then
            call relax2(u,rhs+rho(j)%a)
        else
            call relax2(u,rho(j)%a)
        end if
    end do
end if
END SUBROUTINE mg
END SUBROUTINE mgfas
```

f90 See the discussion after `mglin` on p. 1336 for the changes made in the Fortran 90 versions of the multigrid routines from the Fortran 77 versions.

`TYPE ptr2d...` See discussion after `mglin` on p. 1336.

`RECURSIVE SUBROUTINE mg(j,u,rhs,trerr)` Recall that `mgfas` solves the problem $\mathcal{L}u = 0$, but that nonzero right-hand sides appear during the solution. We implement this by having `rhs` be an optional argument to `mg`. On the first call at a given level `j`, the right-hand side is zero and so you just omit it from the calling sequence. On the other hand, the truncation error `trerr` is computed only on the first call at a given level, so it is also an optional argument that does get supplied on the first call:

```
call mg(j,uj,trerr=trerr)
```

The second and subsequent calls at a given level supply `rhs=tau` but omit `trerr`:

```
call mg(j-1,v,tau)
```

Note that we can omit the keyword `rhs` from this call because the variable `tau` appears in the correct order of arguments. However, in the other call above, the keyword `trerr` must be supplied because `rhs` has been omitted.

The example equation that is solved in `mgfas`, equation (19.6.44), is almost linear, and the code is set up so that ρ is supplied as part of the right-hand side instead of pulling it over to the left-hand side. The variable `rho` is visible to `mg` by host association. Note also that the function `lop` does not include `rho`, but that the statement

```
tau=lop(ut)-rstrct(lop(u))
```

is nevertheless correct, since `rho` would cancel out if it were included in `lop`. This feature is also true in the Fortran 77 code.

```
SUBROUTINE relax2(u,rhs)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
```

Red-black Gauss-Seidel relaxation for equation (19.6.44). The current value of the solution `u` is updated, using the right-hand-side function `rhs`. `u` and `rhs` are square arrays of the same odd dimension.

```
INTEGER(I4B) :: n
REAL(DP) :: foh2,h,h2i
REAL(DP) :: res(size(u,1),size(u,1))
n=assert_eq(size(u,1),size(u,2),size(rhs,1),size(rhs,2),'relax2')
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)
foh2=-4.0_dp*h2i
```

First do the even-even and odd-odd squares of the grid, i.e., the red squares of the checkerboard:

```
res(2:n-1:2,2:n-1:2)=h2i*(u(3:n:2,2:n-1:2)+u(1:n-2:2,2:n-1:2)+&
    u(2:n-1:2,3:n:2)+u(2:n-1:2,1:n-2:2)-4.0_dp*u(2:n-1:2,2:n-1:2))&
    +u(2:n-1:2,2:n-1:2)**2-rhs(2:n-1:2,2:n-1:2)
u(2:n-1:2,2:n-1:2)=u(2:n-1:2,2:n-1:2)-res(2:n-1:2,2:n-1:2)/&
    (foh2+2.0_dp*u(2:n-1:2,2:n-1:2))
res(3:n-2:2,3:n-2:2)=h2i*(u(4:n-1:2,3:n-2:2)+u(2:n-3:2,3:n-2:2)+&
    u(3:n-2:2,4:n-1:2)+u(3:n-2:2,2:n-3:2)-4.0_dp*u(3:n-2:2,3:n-2:2))&
    +u(3:n-2:2,3:n-2:2)**2-rhs(3:n-2:2,3:n-2:2)
u(3:n-2:2,3:n-2:2)=u(3:n-2:2,3:n-2:2)-res(3:n-2:2,3:n-2:2)/&
    (foh2+2.0_dp*u(3:n-2:2,3:n-2:2))
```

Now do even-odd and odd-even squares of the grid, i.e., the black squares of the checkerboard:

```
res(3:n-2:2,2:n-1:2)=h2i*(u(4:n-1:2,2:n-1:2)+u(2:n-3:2,2:n-1:2)+&
    u(3:n-2:2,3:n:2)+u(3:n-2:2,1:n-2:2)-4.0_dp*u(3:n-2:2,2:n-1:2))&
    +u(3:n-2:2,2:n-1:2)**2-rhs(3:n-2:2,2:n-1:2)
u(3:n-2:2,2:n-1:2)=u(3:n-2:2,2:n-1:2)-res(3:n-2:2,2:n-1:2)/&
    (foh2+2.0_dp*u(3:n-2:2,2:n-1:2))
res(2:n-1:2,3:n-2:2)=h2i*(u(3:n:2,3:n-2:2)+u(1:n-2:2,3:n-2:2)+&
    u(2:n-1:2,4:n-1:2)+u(2:n-1:2,2:n-3:2)-4.0_dp*u(2:n-1:2,3:n-2:2))&
    +u(2:n-1:2,3:n-2:2)**2-rhs(2:n-1:2,3:n-2:2)
u(2:n-1:2,3:n-2:2)=u(2:n-1:2,3:n-2:2)-res(2:n-1:2,3:n-2:2)/&
    (foh2+2.0_dp*u(2:n-1:2,3:n-2:2))
END SUBROUTINE relax2
```

f90 See the discussion of red-black relaxation after `sor` on p. 1333.

```
SUBROUTINE slvsm2(u,rhs)
USE nrtype
IMPLICIT NONE
REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
```

Solution of equation (19.6.44) on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in `rhs(1:3,1:3)` and the solution is returned in `u(1:3,1:3)`.

```
REAL(DP) :: disc,fact,h
u=0.0
h=0.5_dp
fact=2.0_dp/h**2
disc=sqrt(fact**2+rhs(2,2))
u(2,2)=-rhs(2,2)/(fact+disc)
END SUBROUTINE slvsm2
```

```
FUNCTION lop(u)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(DP), DIMENSION(:,:), INTENT(IN) :: u
REAL(DP), DIMENSION(size(u,1),size(u,1)) :: lop
```

Given `u`, returns $\mathcal{L}_h(\widetilde{u}_h)$ for equation (19.6.44). `u` and `lop` are square arrays of the same odd dimension.

```
INTEGER(I4B) :: n
REAL(DP) :: h,h2i
n=assert_eq(size(u,1),size(u,2),'lop')
h=1.0_dp/(n-1)
h2i=1.0_dp/(h*h)
lop(2:n-1,2:n-1)=h2i*(u(3:n,2:n-1)+u(1:n-2,2:n-1)+u(2:n-1,3:n)+&
    u(2:n-1,1:n-2)-4.0_dp*u(2:n-1,2:n-1))+u(2:n-1,2:n-1)**2    Interior points.
lop(1:n,1)=0.0                                                 Boundary points.
lop(1:n,n)=0.0
lop(1,1:n)=0.0
lop(n,1:n)=0.0
END FUNCTION lop
```

Chapter B20. Less-Numerical Algorithms

f90 Volume 1's Fortran 77 routine machar performed various clever contortions (due to Cody, Malcolm, and others) to discover the underlying properties of a machine's floating-point representation. Fortran 90, by contrast, provides a built-in set of "numeric inquiry functions" that accomplish the same goal. The routine machar included here makes use of these and is included largely for compatibility with the previous version.

```
SUBROUTINE machar(ibeta,it,irnd,ngrd,machep,negep,iexp,minexp,&
    maxexp,eps,epsneg,xmin,xmax)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ibeta,iexp,irnd,it,machep,maxexp,minexp,negep,ngrd
REAL(SP), INTENT(OUT) :: eps,epsneg,xmax,xmin
REAL(SP), PARAMETER :: RX=1.0
```

Determines and returns machine-specific parameters affecting floating-point arithmetic. Returned values include ibeta, the floating-point radix; it, the number of base-ibeta digits in the floating-point mantissa; eps, the smallest positive number that, added to 1.0, is not equal to 1.0; epsneg, the smallest positive number that, subtracted from 1.0, is not equal to 1.0; xmin, the smallest representable positive number; and xmax, the largest representable positive number. See text for description of other returned parameters. Change all REAL(SP) declarations to REAL(DP) to find double-precision parameters.

```
REAL(SP) :: a,beta,betah,one,temp,tempa,two,zero
ibeta=radix(RX)                          Most of the parameters are easily determined
it=digits(RX)                                from intrinsic functions.
machep=exponent(nearest(RX,RX)-RX)-1
negep=exponent(nearest(RX,-RX)-RX)-1
minexp=minexponent(RX)-1
maxexp=maxexponent(RX)
iexp=nint(log(real(maxexp-minexp+2,sp))/log(2.0_sp))
eps=real(ibeta,sp)**machep
epsneg=real(ibeta,sp)**negep
xmax=huge(RX)
xmin=tiny(RX)
one=RX                                   Determine irnd.
two=one+one
zero=one-one
beta=real(ibeta,sp)
a=beta**(-negep)
irnd=0
betah=beta/two
temp=a+betah
if (temp-a /= zero) irnd=1
tempa=a+beta
temp=tempa+betah
if ((irnd == 0) .and. (temp-tempa /= zero)) irnd=2
ngrd=0                                   Determine ngrd.
```

```
temp=one+eps
if ((irnd == 0) .and. (temp*one-one /= zero)) ngrd=1
temp=xmin/two
if (temp /= zero) irnd=irnd+3          Adjust irnd to reflect partial underflow.
END SUBROUTINE machar
```

⋆ ⋆ ⋆

```
FUNCTION igray(n,is)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n,is
INTEGER(I4B) :: igray
    For zero or positive values of is, return the Gray code of n; if is is negative, return the
    inverse Gray code of n.
INTEGER(I4B) :: idiv,ish
if (is >= 0) then            This is the easy direction!
    igray=ieor(n,n/2)
else                         This is the more complicated direction: In hierarchical stages,
    ish=-1                       starting with a one-bit right shift, cause each bit to be
    igray=n                      XORed with all more significant bits.
    do
        idiv=ishft(igray,ish)
        igray=ieor(igray,idiv)
        if (idiv <= 1 .or. ish == -16) RETURN
        ish=ish+ish          Double the amount of shift on the next cycle.
    end do
end if
END FUNCTION igray
```

⋆ ⋆ ⋆

```
FUNCTION icrc(crc,buf,jinit,jrev)
USE nrtype
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(IN) :: buf
INTEGER(I2B), INTENT(IN) :: crc,jinit
INTEGER(I4B), INTENT(IN) :: jrev
INTEGER(I2B) :: icrc
    Computes a 16-bit Cyclic Redundancy Check for an array buf of bytes, using any of several
    conventions as determined by the settings of jinit and jrev (see accompanying table).
    The result is returned both as an integer icrc and as a 2-byte array crc. If jinit is neg-
    ative, then crc is used on input to initialize the remainder register, in effect concatenating
    buf to the previous call.
INTEGER(I4B), SAVE :: init=0
INTEGER(I2B) :: j,cword,ich
INTEGER(I2B), DIMENSION(0:255), SAVE :: icrctb,rchr
INTEGER(I2B), DIMENSION(0:15) :: it = &      Table of 4-bit bit-reverses.
    (/ 0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15 /)
if (init == 0) then                          Do we need to initialize tables?
    init=1
    do j=0,255                               The two tables are: CRCs of all characters,
        icrctb(j)=icrc1(ishft(j,8),char(0))      and bit-reverses of all characters.
        rchr(j)=ishft(it(iand(j,15_I2B)),4)+it(ishft(j,-4))
    end do
end if
cword=crc
if (jinit >= 0) then                         Initialize the remainder register.
    cword=ior(jinit,ishft(jinit,8))
```

```
else if (jrev < 0) then                    If not initializing, do we reverse the register?
    cword=ior(rchr(hibyte()),ishft(rchr(lobyte()),8))
end if
do j=1,size(buf)                           Main loop over the characters in the array.
    ich=ichar(buf(j))
    if (jrev < 0) ich=rchr(ich)
    cword=ieor(icrctb(ieor(ich,hibyte())),ishft(lobyte(),8))
end do
icrc=merge(cword, &                        Do we need to reverse the output?
    ior(rchr(hibyte()),ishft(rchr(lobyte()),8)), jrev >= 0)
CONTAINS

FUNCTION hibyte()
INTEGER(I2B) :: hibyte
  Extracts the high byte of the 2-byte integer cword.
hibyte = ishft(cword,-8)
END FUNCTION hibyte

FUNCTION lobyte()
INTEGER(I2B) :: lobyte
  Extracts the low byte of the 2-byte integer cword.
lobyte = iand(cword,255_I2B)
END FUNCTION lobyte

FUNCTION icrc1(crc,onech)
INTEGER(I2B), INTENT(IN) :: crc
CHARACTER(1), INTENT(IN) :: onech
INTEGER(I2B) :: icrc1
  Given a remainder up to now, return the new CRC after one character is added. This routine is
  functionally equivalent to icrc(,,-1,1), but slower. It is used by icrc to initialize its table.
INTEGER(I2B) :: i,ich, bit16, ccitt
DATA bit16,ccitt /Z'8000', Z'1021'/
ich=ichar(onech)                           Here is where the character is folded into the
icrc1=ieor(crc,ishft(ich,8))                   register.
do i=1,8                                   Here is where 8 one-bit shifts, and some XORs
    icrc1=merge(ieor(ccitt,ishft(icrc1,1)), &      with the generator polynomial,
        ishft(icrc1,1), iand(icrc1,bit16) /= 0)    are done.
end do
END FUNCTION icrc1
END FUNCTION icrc
```

The embedded functions `hibyte` and `lobyte` always act on the same variable, `cword`. Thus they don't need any explicit argument.

⋆ ⋆ ⋆

```
FUNCTION decchk(string,ch)
USE nrtype; USE nrutil, ONLY : ifirstloc
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(IN) :: string
CHARACTER(1), INTENT(OUT) :: ch
LOGICAL(LGT) :: decchk
```

Decimal check digit computation or verification. Returns as `ch` a check digit for appending to `string`. In this mode, ignore the returned logical value. If `string` already ends with a check digit, returns the function value `.true.` if the check digit is valid, otherwise `.false.` In this mode, ignore the returned value of `ch`. Note that `string` and `ch` contain ASCII characters corresponding to the digits 0-9, *not* byte values in that range. Other ASCII characters are allowed in `string`, and are ignored in calculating the check digit.

```
INTEGER(I4B) :: i,j,k,m
INTEGER(I4B) :: ip(0:9,0:7) = reshape((/ &      Group multiplication and permuta-
    0,1,2,3,4,5,6,7,8,9,1,5,7,6,2,8,3,0,9,4,& tion tables.
    5,8,0,3,7,9,6,1,4,2,8,9,1,6,0,4,3,5,2,7,9,4,5,3,1,2,6,8,7,0,&
    4,2,8,6,5,7,3,9,0,1,2,7,9,3,8,0,6,4,1,5,7,0,4,6,9,1,3,2,5,8 /),&
```

```
    (/ 10,8 /) )
INTEGER(I4B) :: ij(0:9,0:9) = reshape((/ &
    0,1,2,3,4,5,6,7,8,9,1,2,3,4,0,9,5,6,7,8,2,3,4,0,1,8,9,5,6,&
    7,3,4,0,1,2,7,8,9,5,6,4,0,1,2,3,6,7,8,9,5,5,6,7,8,9,0,1,2,3,&
    4,6,7,8,9,5,4,0,1,2,3,7,8,9,5,6,3,4,0,1,2,8,9,5,6,7,2,3,4,0,&
    1,9,5,6,7,8,1,2,3,4,0 /),(/ 10,10 /))
k=0
m=0
do j=1,size(string)                          Look at successive characters.
    i=ichar(string(j))
    if (i >= 48 .and. i <= 57) then          Ignore everything except digits.
        k=ij(k,ip(mod(i+2,10),mod(m,8)))
        m=m+1
    end if
end do
decchk=logical(k == 0,kind=lgt)
i=mod(m,8)                                   Find which appended digit will check prop-
i=ifirstloc(ij(k,ip(0:9,i)) == 0)-1              erly.
ch=char(i+48)                                Convert to ASCII.
END FUNCTION decchk
```

f90 Note the use of the utility function `ifirstloc` to find the first (in this case, the only) correct check digit.

⋆ ⋆ ⋆

f90 The Huffman and arithmetic coding routines exemplify the use of modules to encapsulate user-defined data types. In these algorithms, "the code" is a fairly complicated construct containing scalar and array data. We define types `huffcode` and `arithcode`, then can pass "the code" from the routine that constructs it to the routine that uses it as a single variable.

```
MODULE huf_info
USE nrtype
IMPLICIT NONE
TYPE huffcode
    INTEGER(I4B) :: nch,nodemax
    INTEGER(I4B), DIMENSION(:), POINTER :: icode,left,iright,ncode
END TYPE huffcode
CONTAINS
SUBROUTINE huff_allocate(hcode,mc)
USE nrtype
IMPLICIT NONE
TYPE(huffcode) :: hcode
INTEGER(I4B) :: mc
INTEGER(I4B) :: mq
mq=2*mc-1
allocate(hcode%icode(mq),hcode%ncode(mq),hcode%left(mq),hcode%iright(mq))
hcode%icode(:)=0
hcode%ncode(:)=0
END SUBROUTINE huff_allocate

SUBROUTINE huff_deallocate(hcode)
USE nrtype
IMPLICIT NONE
TYPE(huffcode) :: hcode
deallocate(hcode%iright,hcode%left,hcode%ncode,hcode%icode)
nullify(hcode%icode)
nullify(hcode%ncode)
nullify(hcode%left)
nullify(hcode%iright)
```

```
END SUBROUTINE huff_deallocate
END MODULE huf_info


SUBROUTINE hufmak(nfreq,ilong,nlong,hcode)
USE nrtype; USE nrutil, ONLY : array_copy,arth,imaxloc,nrerror
USE huf_info
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ilong,nlong
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nfreq
TYPE(huffcode) :: hcode
    Given the frequency of occurrence table nfreq of size(nfreq) characters, return the
    Huffman code hcode. Returned values ilong and nlong are the character number that
    produced the longest code symbol, and the length of that symbol.
INTEGER(I4B) :: ibit,j,k,n,node,nused,nerr
INTEGER(I4B), DIMENSION(2*size(nfreq)-1) :: indx,iup,nprob
hcode%nch=size(nfreq)           Initialization.
call huff_allocate(hcode,size(nfreq))
nused=0
nprob(1:hcode%nch)=nfreq(1:hcode%nch)
call array_copy(pack(arth(1,1,hcode%nch), nfreq(1:hcode%nch) /= 0 ),&
    indx,nused,nerr)
do j=nused,1,-1                 Sort nprob into a heap structure in indx.
    call hufapp(j)
end do
k=hcode%nch
do                              Combine heap nodes, remaking the heap at each stage.
    if (nused <= 1) exit
    node=indx(1)
    indx(1)=indx(nused)
    nused=nused-1
    call hufapp(1)
    k=k+1
    nprob(k)=nprob(indx(1))+nprob(node)
    hcode%left(k)=node          Store left and right children of a node.
    hcode%iright(k)=indx(1)
    iup(indx(1))=-k             Indicate whether a node is a left or right child of its par-
    iup(node)=k                     ent.
    indx(1)=k
    call hufapp(1)
end do
hcode%nodemax=k
iup(hcode%nodemax)=0
do j=1,hcode%nch                Make the Huffman code from the tree.
    if (nprob(j) /= 0) then
        n=0
        ibit=0
        node=iup(j)
        do
            if (node == 0) exit
            if (node < 0) then
                n=ibset(n,ibit)
                node=-node
            end if
            node=iup(node)
            ibit=ibit+1
        end do
        hcode%icode(j)=n
        hcode%ncode(j)=ibit
    end if
end do
ilong=imaxloc(hcode%ncode(1:hcode%nch))
nlong=hcode%ncode(ilong)
```

```
if (nlong > bit_size(1_i4b)) call &        Check nlong not larger than word length.
    nrerror('hufmak: Number of possible bits for code exceeded')
CONTAINS
SUBROUTINE hufapp(l)
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: l
  Used by hufmak to maintain a heap structure in the array indx(1:l).
INTEGER(I4B) :: i,j,k,n
n=nused
i=l
k=indx(i)
do
    if (i > n/2) exit
    j=i+i
    if (j < n .and. nprob(indx(j)) > nprob(indx(j+1))) &
        j=j+1
    if (nprob(k) <= nprob(indx(j))) exit
    indx(i)=indx(j)
    i=j
end do
indx(i)=k
END SUBROUTINE hufapp
END SUBROUTINE hufmak
```

```
SUBROUTINE hufenc(ich,codep,nb,hcode)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE huf_info
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: ich
INTEGER(I4B), INTENT(INOUT) :: nb
CHARACTER(1), DIMENSION(:), POINTER :: codep
TYPE(huffcode) :: hcode
    Huffman encode the single character ich (in the range 0..nch-1) using the code in hcode,
    write the result to the character array pointed to by codep starting at bit nb (whose smallest
    valid value is zero), and increment nb appropriately. This routine is called repeatedly to
    encode consecutive characters in a message, but must be preceded by a single initializing
    call to hufmak.
INTEGER(I4B) :: k,l,n,nc,ntmp
k=ich+1                                    Convert character range 0..nch-1 to ar-
if (k > hcode%nch .or. k < 1) call &           ray index range 1..nch.
    nrerror('hufenc: ich out of range')
do n=hcode%ncode(k),1,-1                   Loop over the bits in the stored Huffman
    nc=nb/8+1                                  code for ich.
    if (nc > size(codep)) codep=>reallocate(codep,2*size(codep))
    l=mod(nb,8)
    if (l == 0) codep(nc)=char(0)
    if (btest(hcode%icode(k),n-1)) then    Set appropriate bits in codep.
        ntmp=ibset(ichar(codep(nc)),l)
        codep(nc)=char(ntmp)
    end if
    nb=nb+1
end do
END SUBROUTINE hufenc
```

```
SUBROUTINE hufdec(ich,code,nb,hcode)
USE nrtype
USE huf_info
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: ich
INTEGER(I4B), INTENT(INOUT) :: nb
CHARACTER(1), DIMENSION(:), INTENT(IN) :: code
TYPE(huffcode) :: hcode
```

Starting at bit number nb in the character array code, use the Huffman code in hcode to decode a single character (returned as ich in the range 0..nch−1) and increment nb appropriately. Repeated calls, starting with $\mathtt{nb} = 0$, will return successive characters in a compressed message. The returned value ich=nch indicates end-of-message. This routine must be preceded by a single initializing call to hufmak.

```
INTEGER(I4B) :: l,nc,node
node=hcode%nodemax                          Set node to the top of the decoding tree.
do                                          Loop until a valid character is obtained.
    nc=nb/8+1
    if (nc > size(code)) then               Ran out of input; return with ich=nch
        ich=hcode%nch                           indicating end of message.
        RETURN
    end if
    l=mod(nb,8)                             Now decoding this bit.
    nb=nb+1
    if (btest(ichar(code(nc)),l)) then      Branch left or right in tree, depending on
        node=hcode%iright(node)                 its value.
    else
        node=hcode%left(node)
    end if
    if (node <= hcode%nch) then             If we reach a terminal node, we have a
        ich=node-1                              complete character and can return.
        RETURN
    end if
end do
END SUBROUTINE hufdec
```

⋆ ⋆ ⋆

MODULE arcode_info

```
USE nrtype
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: NWK=20
  NWK is the number of working digits (see text).
TYPE arithcode
    INTEGER(I4B), DIMENSION(:), POINTER :: ilob,iupb,ncumfq
    INTEGER(I4B) :: jdif,nc,minint,nch,ncum,nrad
END TYPE arithcode
```

CONTAINS

```
SUBROUTINE arcode_allocate(acode,mc)
USE nrtype
IMPLICIT NONE
TYPE(arithcode) :: acode
INTEGER(I4B) :: mc
allocate(acode%ilob(NWK),acode%iupb(NWK),acode%ncumfq(mc+2))
END SUBROUTINE arcode_allocate

SUBROUTINE arcode_deallocate(acode)
USE nrtype
IMPLICIT NONE
TYPE(arithcode) :: acode
deallocate(acode%ncumfq,acode%iupb,acode%ilob)
nullify(acode%ilob)
nullify(acode%iupb)
```

```
nullify(acode%ncumfq)
END SUBROUTINE arcode_deallocate
END MODULE arcode_info
```

```
SUBROUTINE arcmak(nfreq,nradd,acode)
USE nrtype; USE nrutil, ONLY : cumsum,nrerror
USE arcode_info
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nradd
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nfreq
TYPE(arithcode) :: acode
INTEGER(I4B), PARAMETER :: MAXINT=huge(nradd)
```

Given a table nfreq of the frequency of occurrence of size(nfreq) symbols, and given a desired output radix nradd, initialize the cumulative frequency table and other variables for arithmetic compression. Store the code in acode.

MAXINT is a large positive integer that does not overflow.

```
if (nradd > 256) call nrerror('output radix may not exceed 256 in arcmak')
acode%minint=MAXINT/nradd
acode%nch=size(nfreq)
acode%nrad=nradd
call arcode_allocate(acode,acode%nch)
acode%ncumfq(1)=0
acode%ncumfq(2:acode%nch+1)=cumsum(max(nfreq(1:acode%nch),1))
acode%ncumfq(acode%nch+2)=acode%ncumfq(acode%nch+1)+1
acode%ncum=acode%ncumfq(acode%nch+2)
END SUBROUTINE arcmak
```

```
SUBROUTINE arcode(ich,codep,lcd,isign,acode)
USE nrtype; USE nrutil, ONLY : nrerror,reallocate
USE arcode_info
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: ich,lcd
INTEGER(I4B), INTENT(IN) :: isign
CHARACTER(1), DIMENSION(:), POINTER :: codep
TYPE(arithcode) :: acode
```

Compress (isign $= 1$) or decompress (isign $= -1$) the single character ich into or out of the character array pointed to by codep, starting with byte codep(lcd) and (if necessary) incrementing lcd so that, on return, lcd points to the first unused byte in codep. Note that this routine saves the result of previous calls until a new byte of code is produced, and only then increments lcd. An initializing call with isign=0 is required for each different array codep. The routine arcmak must have previously been called to initialize the code acode. A call with ich=arcode%nch (as set in arcmak) has the reserved meaning "end of message."

```
INTEGER(I4B) :: ihi,j,ja,jh,jl,m
if (isign == 0) then                          Initialize enough digits of the upper and lower
    acode%jdif=acode%nrad-1                       bounds.
    acode%ilob(:)=0
    acode%iupb(:)=acode%nrad-1
    do j=NWK,1,-1
        acode%nc=j
        if (acode%jdif > acode%minint) RETURN      Initialization complete.
        acode%jdif=(acode%jdif+1)*acode%nrad-1
    end do
    call nrerror('NWK too small in arcode')
else
    if (isign > 0) then                       If encoding, check for valid input character.
        if (ich > acode%nch .or. ich < 0) call nrerror('bad ich in arcode')
    else                                      If decoding, locate the character ich by bi-
        ja=ichar(codep(lcd))-acode%ilob(acode%nc)      section.
        do j=acode%nc+1,NWK
```

```
            ja=ja*acode%nrad+(ichar(codep(j+lcd-acode%nc))-acode%ilob(j))
        end do
        ich=0
        ihi=acode%nch+1
        do
            if (ihi-ich <= 1) exit
            m=(ich+ihi)/2
            if (ja >= jtry(acode%jdif,acode%ncumfq(m+1),acode%ncum)) then
                ich=m
            else
                ihi=m
            end if
        end do
        if (ich == acode%nch) RETURN        Detected end of message.
    end if
      Following code is common for encoding and decoding. Convert character ich to a new
      subrange [ilob,iupb).
    jh=jtry(acode%jdif,acode%ncumfq(ich+2),acode%ncum)
    jl=jtry(acode%jdif,acode%ncumfq(ich+1),acode%ncum)
    acode%jdif=jh-jl
    call arcsum(acode%ilob,acode%iupb,jh,NWK,acode%nrad,acode%nc)
      How many leading digits to output (if encoding) or skip over?
    call arcsum(acode%ilob,acode%ilob,jl,NWK,acode%nrad,acode%nc)
    do j=acode%nc,NWK
        if (ich /= acode%nch .and. acode%iupb(j) /= acode%ilob(j)) exit
        if (acode%nc > size(codep)) codep=>reallocate(codep,2*size(codep))
        if (isign > 0) codep(lcd)=char(acode%ilob(j))
        lcd=lcd+1
    end do
    if (j > NWK) RETURN                     Ran out of message. Did someone forget to
    acode%nc=j                                  encode a terminating ncd?
    j=0                                     How many digits to shift?
    do
        if (acode%jdif >= acode%minint) exit
        j=j+1
        acode%jdif=acode%jdif*acode%nrad
    end do
    if (acode%nc-j < 1) call nrerror('NWK too small in arcode')
    if (j /= 0) then                        Shift them.
        acode%iupb((acode%nc-j):(NWK-j))=acode%iupb(acode%nc:NWK)
        acode%ilob((acode%nc-j):(NWK-j))=acode%ilob(acode%nc:NWK)
    end if
    acode%nc=acode%nc-j
    acode%iupb((NWK-j+1):NWK)=0
    acode%ilob((NWK-j+1):NWK)=0
end if                                      Normal return.
CONTAINS

FUNCTION jtry(m,n,k)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: m,n,k
INTEGER(I4B) :: jtry
  Calculate (m*n)/k without overflow. Program efficiency can be improved by substituting an
  assembly language routine that does integer multiply to a double register.
jtry=int((real(m,dp)*real(n,dp))/real(k,dp))
END FUNCTION jtry

SUBROUTINE arcsum(iin,iout,ja,nwk,nrad,nc)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iin
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: iout
INTEGER(I4B), INTENT(IN) :: nwk,nrad,nc
INTEGER(I4B), INTENT(INOUT) :: ja
```

Add the integer ja to the radix nrad multiple-precision integer iin(nc..nwk). Return the result in iout(nc..nwk).

```
INTEGER(I4B) :: j,jtmp,karry
karry=0
do j=nwk,nc+1,-1
    jtmp=ja
    ja=ja/nrad
    iout(j)=iin(j)+(jtmp-ja*nrad)+karry
    if (iout(j) >= nrad) then
        iout(j)=iout(j)-nrad
        karry=1
    else
        karry=0
    end if
end do
iout(nc)=iin(nc)+ja+karry
END SUBROUTINE arcsum
END SUBROUTINE arcode
```

⋆ ⋆ ⋆

MODULE mpops

```
USE nrtype
INTEGER(I4B), PARAMETER :: NPAR_ICARRY=64
CONTAINS
```

```
SUBROUTINE icarry(karry,isum,nbits)
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: karry
```

Perform deferred carry operation on an array isum of multiple-precision digits. Nonzero bits of higher order than nbits (typically 8) are carried to the next-lower (leftward) component of isum. The final (most leftward) carry value is returned as karry.

```
INTEGER(I2B), DIMENSION(:), INTENT(INOUT) :: isum
INTEGER(I4B), INTENT(IN) :: nbits
INTEGER(I4B) :: n,j
INTEGER(I2B), DIMENSION(size(isum)) :: ihi
INTEGER(I2B) :: mb,ihh
n=size(isum)
mb=ishft(1,nbits)-1                          Make mask for low-order bits.
karry=0
if (n < NPAR_ICARRY ) then
    do j=n,2,-1                              Keep going until all carries have cascaded.
        ihh=ishft(isum(j),-nbits)
        if (ihh /= 0) then
            isum(j)=iand(isum(j),mb)
            isum(j-1)=isum(j-1)+ihh
        end if
    end do
    ihh=ishft(isum(1),-nbits)
    isum(1)=iand(isum(1),mb)
    karry=karry+ihh
else
    do
        ihi=ishft(isum,-nbits)               Get high bits.
        if (all(ihi == 0)) exit              Check if done.
        where (ihi /= 0) isum=iand(isum,mb)      Remove bits to be carried and add
        where (ihi(2:n) /= 0) isum(1:n-1)=isum(1:n-1)+ihi(2:n)     them to left.
        karry=karry+ihi(1)                   Final carry.
    end do
end if
END SUBROUTINE icarry
```

```
SUBROUTINE mpadd(w,u,v,n)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
INTEGER(I4B), INTENT(IN) :: n
```

Adds the unsigned radix 256 integers `u(1:n)` and `v(1:n)` yielding the unsigned integer `w(1:n+1)`.

```
INTEGER(I2B), DIMENSION(n) :: isum
INTEGER(I4B) :: karry
isum=ichar(u(1:n))+ichar(v(1:n))
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpadd
```

```
SUBROUTINE mpsub(is,w,u,v,n)
IMPLICIT NONE
INTEGER(I4B), INTENT(OUT) :: is
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
INTEGER(I4B), INTENT(IN) :: n
```

Subtracts the unsigned radix 256 integer `v(1:n)` from `u(1:n)` yielding the unsigned integer `w(1:n)`. If the result is negative (wraps around), `is` is returned as -1; otherwise it is returned as 0.

```
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=255+ichar(u(1:n))-ichar(v(1:n))
isum(n)=isum(n)+1
call icarry(karry,isum,8_I4B)
w(1:n)=char(isum)
is=karry-1
END SUBROUTINE mpsub
```

```
SUBROUTINE mpsad(w,u,n,iv)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u
INTEGER(I4B), INTENT(IN) :: n,iv
```

Short addition: The integer `iv` (in the range $0 \leq$ `iv` ≤ 255) is added to the unsigned radix 256 integer `u(1:n)`, yielding `w(1:n+1)`.

```
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=ichar(u(1:n))
isum(n)=isum(n)+iv
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpsad
```

```
SUBROUTINE mpsmu(w,u,n,iv)
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u
INTEGER(I4B), INTENT(IN) :: n,iv
```

Short multiplication: The unsigned radix 256 integer `u(1:n)` is multiplied by the integer `iv` (in the range $0 \leq$ `iv` ≤ 255), yielding `w(1:n+1)`.

```
INTEGER(I4B) :: karry
INTEGER(I2B), DIMENSION(n) :: isum
isum=ichar(u(1:n))*iv
call icarry(karry,isum,8_I4B)
w(2:n+1)=char(isum)
w(1)=char(karry)
END SUBROUTINE mpsmu
```

```
SUBROUTINE mpneg(u,n)
IMPLICIT NONE
```

```
    CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: u
    INTEGER(I4B), INTENT(IN) :: n
      Ones-complement negate the unsigned radix 256 integer u(1:n).
    INTEGER(I4B) :: karry
    INTEGER(I2B), DIMENSION(n) :: isum
    isum=255-ichar(u(1:n))
    isum(n)=isum(n)+1
    call icarry(karry,isum,8_I4B)
    u(1:n)=char(isum)
    END SUBROUTINE mpneg

    SUBROUTINE mplsh(u,n)
    IMPLICIT NONE
    CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: u
    INTEGER(I4B), INTENT(IN) :: n
      Left shift u(2..n+1) onto u(1:n).
    u(1:n)=u(2:n+1)
    END SUBROUTINE mplsh

    SUBROUTINE mpmov(u,v,n)
    IMPLICIT NONE
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: u
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
    INTEGER(I4B), INTENT(IN) :: n
      Move v(1:n) onto u(1:n).
    u(1:n)=v(1:n)
    END SUBROUTINE mpmov

    SUBROUTINE mpsdv(w,u,n,iv,ir)
    IMPLICIT NONE
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: u
    INTEGER(I4B), INTENT(IN) :: n,iv
    INTEGER(I4B), INTENT(OUT) :: ir
      Short division: The unsigned radix 256 integer u(1:n) is divided by the integer iv (in the
      range 0 ≤ iv ≤ 255), yielding a quotient w(1:n) and a remainder ir (with 0 ≤ ir ≤ 255).
      Note: Your Numerical Recipes authors don't know how to parallelize this routine in Fortran
      90!
    INTEGER(I4B) :: i,j
    ir=0
    do j=1,n
        i=256*ir+ichar(u(j))
        w(j)=char(i/iv)
        ir=mod(i,iv)
    end do
    END SUBROUTINE mpsdv
    END MODULE mpops



    SUBROUTINE mpmul(w,u,v,n,m)
    USE nrtype; USE nrutil, ONLY : nrerror
    USE nr, ONLY : realft
    IMPLICIT NONE
    INTEGER(I4B), INTENT(IN) :: n,m
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
    CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
!   The logical dimensions are: CHARACTER(1) :: w(n+m),u(n),v(m)
    REAL(DP), PARAMETER :: RX=256.0
      Uses fast Fourier transform to multiply the unsigned radix 256 integers u(1:n) and v(1:m),
      yielding a product w(1:n+m).
    INTEGER(I4B) :: j,mn,nn
    REAL(DP) :: cy,t
    REAL(DP), DIMENSION(:), ALLOCATABLE :: a,b,tb
    mn=max(m,n)
    nn=1                                  Find the smallest useable power of two for the transform.
```

```
do
    if (nn >= mn) exit
    nn=nn+nn
end do
nn=nn+nn
allocate(a(nn),b(nn),tb((nn-1)/2))
a(1:n)=ichar(u(1:n))            Move U to a double-precision floating array.
a(n+1:nn)=0.0
b(1:m)=ichar(v(1:m))            Move V to a double-precision floating array.
b(m+1:nn)=0.0
call realft(a(1:nn),1)          Perform the convolution: First, the two Fourier trans-
call realft(b(1:nn),1)              forms.
b(1)=b(1)*a(1)                  Then multiply the complex results (real and imaginary
b(2)=b(2)*a(2)                      parts).
tb=b(3:nn:2)
b(3:nn:2)=tb*a(3:nn:2)-b(4:nn:2)*a(4:nn:2)
b(4:nn:2)=tb*a(4:nn:2)+b(4:nn:2)*a(3:nn:2)
call realft(b(1:nn),-1)         Then do the inverse Fourier transform.
b(:)=b(:)/(nn/2)
cy=0.0                          Make a final pass to do all the carries.
do j=nn,1,-1
    t=b(j)+cy+0.5_dp            The 0.5 allows for roundoff error.
    b(j)=mod(t,RX)
    cy=int(t/RX)
end do
if (cy >= RX) call nrerror('mpmul: sanity check failed in fftmul')
w(1)=char(int(cy))              Copy answer to output.
w(2:(n+m))=char(int(b(1:(n+m-1))))
deallocate(a,b,tb)
END SUBROUTINE mpmul
```

```
SUBROUTINE mpinv(u,v,n,m)
USE nrtype; USE nrutil, ONLY : poly
USE nr, ONLY : mpmul
USE mpops, ONLY : mpmov,mpneg
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: u
CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B), PARAMETER :: MF=4
REAL(SP), PARAMETER :: BI=1.0_sp/256.0_sp
```

Character string `v(1:m)` is interpreted as a radix 256 number with the radix point after (nonzero) `v(1)`; `u(1:n)` is set to the most significant digits of its reciprocal, with the radix point after `u(1)`.

```
INTEGER(I4B) :: i,j,mm
REAL(SP) :: fu
CHARACTER(1), DIMENSION(:), ALLOCATABLE :: rr,s
allocate(rr(max(n,m)+n+1),s(n))
mm=min(MF,m)
fu=1.0_sp/poly(BI,real(ichar(v(:)),sp))     Use ordinary floating arithmetic to get an
do j=1,n                                        initial approximation.
    i=int(fu)
    u(j)=char(i)
    fu=256.0_sp*(fu-i)
end do
do                                          Iterate Newton's rule to convergence.
    call mpmul(rr,u,v,n,m)                  Construct 2 − UV in S.
    call mpmov(s,rr(2:),n)
    call mpneg(s,n)
    s(1)=char(ichar(s(1))-254)              Multiply SU into U.
    call mpmul(rr,s,u,n,n)
    call mpmov(u,rr(2:),n)
```

```
    if (all(ichar(s(2:n-1)) == 0)) exit        If fractional part of S is not zero, it has
end do                                             not converged to 1.
deallocate(rr,s)
END SUBROUTINE mpinv


SUBROUTINE mpdiv(q,r,u,v,n,m)
USE nrtype; USE nrutil, ONLY : nrerror
USE nr, ONLY : mpinv,mpmul
USE mpops, ONLY : mpsad,mpmov,mpsub
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: q,r
CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
!  The logical dimensions are: CHARACTER(1) :: q(n-m+1),r(m),u(n),v(m)
INTEGER(I4B), INTENT(IN) :: n,m
    Divides unsigned radix 256 integers u(1:n) by v(1:m) (with m ≤ n required), yielding a
    quotient q(1:n-m+1) and a remainder r(1:m).
INTEGER(I4B), PARAMETER :: MACC=6
INTEGER(I4B) :: is
CHARACTER(1), DIMENSION(:), ALLOCATABLE, TARGET :: rr,s
CHARACTER(1), DIMENSION(:), POINTER :: rr2,s3
allocate(rr(2*(n+MACC)),s(2*(n+MACC)))
rr2=>rr(2:)
s3=>s(3:)
call mpinv(s,v,n+MACC,m)              Set S = 1/V.
call mpmul(rr,s,u,n+MACC,n)           Set Q = SU.
call mpsad(s,rr,n+n+MACC/2,1)
call mpmov(q,s3,n-m+1)
call mpmul(rr,q,v,n-m+1,m)            Multiply and subtract to get the remainder.
call mpsub(is,rr2,u,rr2,n)
if (is /= 0) call nrerror('MACC too small in mpdiv')
call mpmov(r,rr(n-m+2:),m)
deallocate(rr,s)
END SUBROUTINE mpdiv


SUBROUTINE mpsqrt(w,u,v,n,m)
USE nrtype; USE nrutil, ONLY : poly
USE nr, ONLY : mpmul
USE mpops, ONLY : mplsh,mpmov,mpneg,mpsdv
IMPLICIT NONE
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w,u
CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
INTEGER(I4B), INTENT(IN) :: n,m
INTEGER(I4B), PARAMETER :: MF=3
REAL(SP), PARAMETER :: BI=1.0_sp/256.0_sp
    Character string v(1:m) is interpreted as a radix 256 number with the radix point after
    v(1); w(1:n) is set to its square root (radix point after w(1)), and u(1:n) is set to the
    reciprocal thereof (radix point before u(1)). w and u need not be distinct, in which case
    they are set to the square root.
INTEGER(I4B) :: i,ir,j,mm
REAL(SP) :: fu
CHARACTER(1), DIMENSION(:), ALLOCATABLE :: r,s
allocate(r(2*n),s(2*n))
mm=min(m,MF)
fu=1.0_sp/sqrt(poly(BI,real(ichar(v(:)),sp)))      Use ordinary floating arithmetic
do j=1,n                                              to get an initial approxima-
    i=int(fu)                                         tion.
    u(j)=char(i)
    fu=256.0_sp*(fu-i)
end do
do                                                 Iterate Newton's rule to convergence.
    call mpmul(r,u,u,n,n)                          Construct S = (3 − VU^2)/2.
```

```
    call mplsh(r,n)
    call mpmul(s,r,v,n,min(m,n))
    call mplsh(s,n)
    call mpneg(s,n)
    s(1)=char(ichar(s(1))-253)
    call mpsdv(s,s,n,2,ir)
    if (any(ichar(s(2:n-1)) /= 0)) then
          If fractional part of S is not zero, it has not converged to 1.
        call mpmul(r,s,u,n,n)                Replace U by SU.
        call mpmov(u,r(2:),n)
        cycle
    end if
    call mpmul(r,u,v,n,min(m,n))             Get square root from reciprocal and return.
    call mpmov(w,r(2:),n)
    deallocate(r,s)
    RETURN
end do
END SUBROUTINE mpsqrt



SUBROUTINE mp2dfr(a,s,n,m)
USE nrtype
USE mpops, ONLY : mplsh,mpsmu
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), INTENT(OUT) :: m
CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: a
CHARACTER(1), DIMENSION(:), INTENT(OUT) :: s
INTEGER(I4B), PARAMETER :: IAZ=48
    Converts a radix 256 fraction a(1:n) (radix point before a(1)) to a decimal fraction
    represented as an ascii string s(1:m), where m is a returned value. The input array a(1:n)
    is destroyed. NOTE: For simplicity, this routine implements a slow (∝ N^2) algorithm. Fast
    (∝ N ln N), more complicated, radix conversion algorithms do exist.
INTEGER(I4B) :: j
m=int(2.408_sp*n)
do j=1,m
    call mpsmu(a,a,n,10)
    s(j)=char(ichar(a(1))+IAZ)
    call mplsh(a,n)
end do
END SUBROUTINE mp2dfr



SUBROUTINE mppi(n)
USE nrtype
USE nr, ONLY : mp2dfr,mpinv,mpmul,mpsqrt
USE mpops, ONLY : mpadd,mplsh,mpmov,mpsdv
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), PARAMETER :: IAOFF=48
    Demonstrate multiple precision routines by calculating and printing the first n bytes of π.
INTEGER(I4B) :: ir,j,m
CHARACTER(1), DIMENSION(n) :: sx,sxi
CHARACTER(1), DIMENSION(2*n) :: t,y
CHARACTER(1), DIMENSION(3*n) :: s
CHARACTER(1), DIMENSION(n+1) :: x,bigpi
t(1)=char(2)                                 Set T = 2.
t(2:n)=char(0)
call mpsqrt(x,x,t,n,n)                       Set X_0 = √2.
call mpadd(bigpi,t,x,n)                      Set π_0 = 2 + √2.
call mplsh(bigpi,n)
call mpsqrt(sx,sxi,x,n,n)                    Set Y_0 = 2^{1/4}.
```

```
call mpmov(y,sx,n)
do
    call mpadd(x,sx,sxi,n)                Set X_{i+1} = (X_i^{1/2} + X_i^{-1/2})/2.
    call mpsdv(x,x(2:),n,2,ir)
    call mpsqrt(sx,sxi,x,n,n)             Form the temporary T = Y_i X_{i+1}^{1/2} + X_{i+1}^{-1/2}.
    call mpmul(t,y,sx,n,n)
    call mpadd(t(2:),t(2:),sxi,n)
    x(1)=char(ichar(x(1))+1)              Increment X_{i+1} and Y_i by 1.
    y(1)=char(ichar(y(1))+1)
    call mpinv(s,y,n,n)                   Set Y_{i+1} = T/(Y_i + 1).
    call mpmul(y,t(3:),s,n,n)
    call mplsh(y,n)
    call mpmul(t,x,s,n,n)                 Form temporary T = (X_{i+1} + 1)/(Y_i + 1).
    m=mod(255+ichar(t(2)),256)            If T = 1 then we have converged.
    if (abs(ichar(t(n+1))-m) > 1 .or. any(ichar(t(3:n)) /= m)) then
        call mpmul(s,bigpi,t(2:),n,n)     Set π_{i+1} = T π_i.
        call mpmov(bigpi,s(2:),n)
        cycle
    end if
    write (*,*) 'pi='
    s(1)=char(ichar(bigpi(1))+IAOFF)
    s(2)='.'
    call mp2dfr(bigpi(2:),s(3:),n-1,m)
      Convert to decimal for printing. NOTE: The conversion routine, for this demonstration
      only, is a slow (∝ N^2) algorithm. Fast (∝ N ln N), more complicated, radix conversion
      algorithms do exist.
    write (*,'(1x,64a1)') (s(j),j=1,m+1)
    RETURN
end do
END SUBROUTINE mppi
```

Numerical Recipes
in Fortran 90
Second Edition

Volume 2 of
Fortran Numerical Recipes

Numerical Recipes
in Fortran 90
The Art of *Parallel* Scientific Computing
Second Edition

Volume 2 of
Fortran Numerical Recipes

William H. Press

Harvard-Smithsonian Center for Astrophysics

Saul A. Teukolsky

Department of Physics, Cornell University

William T. Vetterling

Polaroid Corporation

Brian P. Flannery

EXXON Research and Engineering Company

Foreword by

Michael Metcalf

CERN, Geneva, Switzerland

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

Copyright © Cambridge University Press 1986, 1996,
except for all computer programs and procedures, which are
Copyright © Numerical Recipes Software 1986, 1996,
and except for Appendix C1, which is placed into the public domain.
All Rights Reserved.

Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing,
Volume 2 of Fortran Numerical Recipes, Second Edition, first published 1996.
Reprinted with corrections 1997.
The code in this volume is corrected to software version 2.08

Printed in the United States of America
Typeset in TEX

Without an additional license to use the contained software, this book is intended as a text and reference book, for reading purposes only. A free license for limited use of the software by the individual owner of a copy of this book who personally types one or more routines into a single computer is granted under terms described on p. xviii. See the section "License Information" (pp. xvii–xx) for information on obtaining more general licenses at low cost.

Machine-readable media containing the software in this book, with included licenses for use on a single screen, are available from Cambridge University Press. See the order form at the back of the book, email to "orders@cup.org" (North America) or "trade@cup.cam.ac.uk" (rest of world), or write to Cambridge University Press, 110 Midland Avenue, Port Chester, NY 10573 (USA), for further information.

The software may also be downloaded, with immediate purchase of a license also possible, from the Numerical Recipes Software Web site (`http://www.nr.com`). Unlicensed transfer of Numerical Recipes programs to any other format, or to any computer except one that is specifically licensed, is strictly prohibited. Technical questions, corrections, and requests for information should be addressed to Numerical Recipes Software, P.O. Box 243, Cambridge, MA 02238 (USA), email "info@nr.com", or fax 781 863-1739.

Library of Congress Cataloging-in-Publication Data

Numerical recipes in Fortran 90 : the art of parallel scientific computing / William H. Press ... [et al.]. – 2nd ed.
p. cm.

Includes bibliographical references and index.

ISBN 0-521-57439-0 (hardcover)

1. FORTRAN 90 (Computer program language) 2. Parallel programming (Computer science) 3. Numerical analysis–Data processing.

I. Press, William H.
QA76.73.F25N85 1996
519.4′0285′52–dc20 96-5567
CIP

A catalog record for this book is available from the British Library.

ISBN 0 521 57439 0 Volume 2 (this book)
ISBN 0 521 43064 X Volume 1
ISBN 0 521 43721 0 Example book in FORTRAN
ISBN 0 521 57440 4 FORTRAN diskette (IBM 3.5″)
ISBN 0 521 57608 3 CDROM (IBM PC/Macintosh)
ISBN 0 521 57607 5 CDROM (UNIX)

Contents

Preface to Volume 2 ***viii***

Foreword by Michael Metcalf ***x***

License Information ***xvii***

21 Introduction to Fortran 90 Language Features ***935***

21.0 Introduction 935
21.1 Quick Start: Using the Fortran 90 Numerical Recipes Routines 936
21.2 Fortran 90 Language Concepts 937
21.3 More on Arrays and Array Sections 941
21.4 Fortran 90 Intrinsic Procedures 945
21.5 Advanced Fortran 90 Topics 953
21.6 And Coming Soon: Fortran 95 959

22 Introduction to Parallel Programming ***962***

22.0 Why Think Parallel? 962
22.1 Fortran 90 Data Parallelism: Arrays and Intrinsics 964
22.2 Linear Recurrence and Related Calculations 971
22.3 Parallel Synthetic Division and Related Algorithms 977
22.4 Fast Fourier Transforms 981
22.5 Missing Language Features 983

23 Numerical Recipes Utility Functions for Fortran 90 ***987***

23.0 Introduction and Summary Listing 987
23.1 Routines That Move Data 990
23.2 Routines Returning a Location 992
23.3 Argument Checking and Error Handling 994
23.4 Routines for Polynomials and Recurrences 996
23.5 Routines for Outer Operations on Vectors 1000
23.6 Routines for Scatter with Combine 1002
23.7 Routines for Skew Operations on Matrices 1004
23.8 Other Routines 1007

Fortran 90 Code Chapters ***1009***

B1 Preliminaries ***1010***

B2 Solution of Linear Algebraic Equations ***1014***

B3 Interpolation and Extrapolation ***1043***

B4 Integration of Functions 1052
B5 Evaluation of Functions 1070
B6 Special Functions 1083
B7 Random Numbers 1141
B8 Sorting 1167
B9 Root Finding and Nonlinear Sets of Equations 1182
B10 Minimization or Maximization of Functions 1201
B11 Eigensystems 1225
B12 Fast Fourier Transform 1235
B13 Fourier and Spectral Applications 1253
B14 Statistical Description of Data 1269
B15 Modeling of Data 1285
B16 Integration of Ordinary Differential Equations 1297
B17 Two Point Boundary Value Problems 1314
B18 Integral Equations and Inverse Theory 1325
B19 Partial Differential Equations 1332
B20 Less-Numerical Algorithms 1343

References 1359

Appendices

C1 Listing of Utility Modules (nrtype and nrutil) 1361
C2 Alphabetical Listing of Explicit Interfaces 1384
C3 Index of Programs and Dependencies 1434

General Index to Volumes 1 and 2 1447

Contents of Volume 1: Numerical Recipes in Fortran 77

	Plan of the Two-Volume Edition	xiii
	Preface to the Second Edition	xv
	Preface to the First Edition	xviii
	License Information	xx
	Computer Programs by Chapter and Section	xxiv
1	Preliminaries	1
2	Solution of Linear Algebraic Equations	22
3	Interpolation and Extrapolation	99
4	Integration of Functions	123
5	Evaluation of Functions	159
6	Special Functions	205
7	Random Numbers	266
8	Sorting	320
9	Root Finding and Nonlinear Sets of Equations	340
10	Minimization or Maximization of Functions	387
11	Eigensystems	449
12	Fast Fourier Transform	490
13	Fourier and Spectral Applications	530
14	Statistical Description of Data	603
15	Modeling of Data	650
16	Integration of Ordinary Differential Equations	701
17	Two Point Boundary Value Problems	745
18	Integral Equations and Inverse Theory	779
19	Partial Differential Equations	818
20	Less-Numerical Algorithms	881
	References	916
	Index of Programs and Dependencies	921

Preface to Volume 2

Fortran 90 is not just the long-awaited updating of the Fortran language to modern computing practices. It is also the vanguard of a much larger revolution in computing, that of multiprocessor computers and widespread parallel programming. Parallel computing has been a feature of the largest supercomputers for quite some time. Now, however, it is rapidly moving towards the desktop.

As we watched the gestation and birth of Fortran 90 by its governing "X3J3 Committee" (a process interestingly described by a leading committee member, Michael Metcalf, in the Foreword that follows), it became clear to us that the right moment for moving Numerical Recipes from Fortran 77 to Fortran 90 was sooner, rather than later.

Fortran 90 compilers are now widely available. Microsoft's Fortran PowerStation for Windows 95 brings that firm's undeniable marketing force to PC desktop; we have tested this compiler thoroughly on our code and found it excellent in compatibility and performance. In the UNIX world, we have similarly tested, and had generally fine experiences with, DEC's Fortran 90 for Alpha AXP and IBM's xlf for RS/6000 and similar machines. NAG's Fortran 90 compiler also brings excellent Fortran 90 compatibility to a variety of UNIX platforms. There are no doubt other excellent compilers, both available and on the way. Fortran 90 is completely backwards compatible with Fortran 77, by the way, so you don't have to throw away your legacy code, or keep an old compiler around.

There have been previous special versions of Fortran for parallel supercomputers, but always specific to a particular hardware. Fortran 90, by contrast, is designed to provide a general, architecture-independent framework for parallel computation. Equally importantly, it is an international standard, agreed upon by a large group of computer hardware and software manufacturers and international standards bodies.

With the Fortran 90 language as a tool, we want this volume to be your complete guide for learning how to "think parallel." The language itself is very general in this regard, and applicable to many present and future computers, or even to other parallel computing languages as they come along. Our treatment emphasizes general

principles, but we are also not shy about pointing out parallelization "tricks" that have frequent applicability. These are not only discussed in this volume's principal text chapters (Chapters 21–23), but are also sprinkled throughout the chapters of Fortran 90 code, called out by a special "parallel

hint" logo (left, above). Also scattered throughout the code chapters are specific "Fortran 90 tips," with their own distinct graphic call-out (left). After you read the text chapters, you might want simply to browse among these hints and tips.

A special note to C programmers: Right now, there is no effort at producing a parallel version of C that is comparable to Fortran 90 in maturity, acceptance, and stability. We think, therefore, that C programmers will be well served by using this volume for an educational excursion into Fortran 90, its parallel programming constructions, and the numerical algorithms that capitalize on them. C and C++ programming have not been far from our minds as we have written this volume, and we think that you will find that time spent in absorbing its principal lessons (in Chapters 21–23) will be amply repaid in the future, as C and C++ eventually develop standard parallel extensions.

A final word of truth in packaging: **Don't buy this volume unless you also buy (or already have) Volume 1** (now retitled *Numerical Recipes in Fortran 77*). Volume 2 does not repeat any of the discussion of what individual programs actually do, or of the mathematical methods they utilize, or how to use them. While our Fortran 90 code is thoroughly commented, and includes a header comment for each routine that describes its input and output quantities, these comments are *not* supposed to be a complete description of the programs; the complete descriptions are in Volume 1, which we reference frequently. But here's a money-saving hint to our previous readers: If you already own a Second Edition version whose title is *Numerical Recipes in FORTRAN* (which doesn't indicate either "Volume 1" or "Volume 2" on its title page) then take a marking pen and write in the words "Volume 1." There! (Differences between the previous reprintings and the newest reprinting, the one labeled "Volume 1," are minor.)

Acknowledgments

We continue to be in the debt of many colleagues who give us the benefit of their numerical and computational experience. Many, though not all, of these are listed by name in the preface to the second edition, in Volume 1. To that list we must now certainly add George Marsaglia, whose ideas have greatly influenced our new discussion of random numbers in this volume (Chapter B7).

With this volume, we must acknowledge our additional gratitude and debt to a number of people who generously provided advice, expertise, and time (a great deal of time, in some cases) in the areas of parallel programming and Fortran 90. The original inspiration for this volume came from Mike Metcalf, whose clear lectures on Fortran 90 (in this case, overlooking the beautiful Adriatic at Trieste) convinced us that Fortran 90 could serve as the vehicle for a book with the larger scope of parallel programming generally, and whose continuing advice throughout the project has been indispensable. Gyan Bhanot also played a vital early role in the development of this book; his first translations of our Fortran 77 programs taught us a lot about parallel programming. We are also grateful to Greg Lindhorst, Charles Van Loan, Amos Yahil, Keith Kimball, Malcolm Cohen, Barry Caplin, Loren Meissner, Mitsu Sakamoto, and George Schnurer for helpful correspondence and/or discussion of Fortran 90's subtler aspects.

We once again express in the strongest terms our gratitude to programming consultant Seth Finkelstein, whose contribution to both the coding and the thorough testing of all the routines in this book (against multiple compilers and in sometimes-buggy, and always challenging, early versions) cannot be overstated.

WHP and SAT acknowledge the continued support of the U.S. National Science Foundation for their research on computational methods.

February 1996

William H. Press
Saul A. Teukolsky
William T. Vetterling
Brian P. Flannery

Foreword

by Michael Metcalf

Sipping coffee on a sunbaked terrace can be surprisingly productive. One of the *Numerical Recipes* authors and I were each lecturing at the International Center for Theoretical Physics in Trieste, Italy, he on numerical analysis and I on Fortran 90. The numerical analysis community had made important contributions to the development of the new Fortran standard, and so, unsurprisingly, it became quickly apparent that the algorithms for which *Numerical Recipes* had become renowned could, to great advantage, be recast in a new mold. These algorithms had, hitherto, been expressed in serial form, first in Fortran 77 and then in C, Pascal, and Basic. Now, nested iterations could be replaced by array operations and assignments, and the other features of a rich array language could be exploited. Thus was the idea of a "Numerical Recipes in Fortran 90" first conceived and, after three years' gestation, it is a delight to assist at the birth.

But what *is* Fortran 90? How did it begin, what shaped it, and how, after nearly foundering, did its driving forces finally steer it to a successful conclusion?

The Birth of a Standard

Back in 1966, the version of Fortran now known as Fortran 66 was the first language ever to be standardized, by the predecessor of the present American National Standards Institute (ANSI). It was an all-American affair. Fortran had first been developed by John Backus of IBM in New York, and it was the dominant scientific programming language in North America. Many Europeans preferred Algol (in which Backus had also had a hand). Eventually, however, the mathematicians who favored Algol for its precisely expressible syntax began to defer to the scientists and engineers who appreciated Fortran's pragmatic, even natural, style. In 1978, the upgraded Fortran 77 was standardized by the ANSI technical committee, X3J3, and subsequently endorsed by other national bodies and by ISO in Geneva, Switzerland. Its dominance in all fields of scientific and numerical computing grew as new, highly optimizing compilers came onto the market. Although newer languages, particularly Pascal, Basic, PL/1, and later Ada attracted their own adherents, scientific users throughout the 1980s remained true to Fortran. Only towards the end of that decade did C draw increasing support from scientific programmers who had discovered the power of structures and pointers.

During all this time, X3J3 kept functioning, developing the successor version to Fortran 77. It was to be a decade of strife and contention. The early plans, in the late 1970s, were mainly to add to Fortran 77 features that had had to be left out of that standard. Among these were dynamic storage and an array language, enabling it to map directly onto the architecture of supercomputers, then coming onto the market. The intention was to have this new version ready within five years, in 1982. But two new factors became significant at that time. The first was the decision that the next standard should not just codify existing practice, as had largely been the case in 1966 and 1978, but also extend the functionality of the language through

innovative additions (even though, for the array language, there was significant borrowing from John Iverson's APL and from DAP Fortran). The second factor was that X3J3 no longer operated under only American auspices. In the course of the 1980s, the standardization of programming languages came increasingly under the authority of the international body, ISO. Initially this was in an advisory role, but now ISO is the body that, through its technical committee WG5 (in full, ISO/IEC JTC1/SC22/WG5), is responsible for determining the course of the language. WG5 also steers the work of the development body, then as now, the highly skilled and competent X3J3. As we shall see, this shift in authority was crucial at the most difficult moment of Fortran 90's development.

The internationalization of the standards effort was reflected in the welcome given by X3J3 to six or seven European members; they, and about one-third of X3J3's U.S. members, provided the overlapping core of membership of X3J3 and WG5 that was vital in the final years in bringing the work to a successful conclusion. X3J3 membership, which peaked at about 45, is restricted to one voting member per organization, and significant decisions require a majority of two-thirds of those voting. Nationality plays no role, except in determining the U.S. position on an international issue. Members, who are drawn mainly from the vendors, large research laboratories, and academia, must be present or represented at two-thirds of all meetings in order to retain voting rights.

In 1980, X3J3 reported on its plans to the forerunner of WG5 in Amsterdam, Holland. Fortran 8x, as it was dubbed, was to have a basic array language, new looping constructs, a bit data type, data structures, a free source form, a mechanism to "group" procedures, and another to manage the global name space. Old features, including `COMMON`, `EQUIVALENCE`, and the arithmetic-`IF`, were to be consigned to a so-called obsolete module, destined to disappear in a subsequent revision. This was part of the "core plus modules" architecture, for adding new features and retiring old ones, an aid to backwards compatibility. Even though Fortran 77 compilers were barely available, the work seemed well advanced and the mood was optimistic. Publication was intended to take place in 1985. It was not to be.

One problem was the sheer number of new features that were proposed as additions to the language, most of them worthwhile in themselves but with the totality being too large. This became a recurrent theme throughout the development of the standard. One example was the suggestion of Lawrie Schonfelder (Liverpool University), at a WG5 meeting in Vienna, Austria, in 1982, that certain features already proposed as additions could be combined to provide a full-blown derived data type facility, thus providing Fortran with abstract data types. This idea was taken up by X3J3 and has since come to be recognized, along with the array language, as one of the two main advances brought about by what became Fortran 90. However, the ramifications go very deep: all the technical details of how to handle arrays of objects of derived types that in turn have array components that have the pointer attribute, and so forth, have to be precisely defined and rigorously specified.

Conflict

The meetings of X3J3 were often full of drama. Most compiler vendors were represented as a matter of course but, for many, their main objective appeared to be to maintain the status quo and to ensure that Fortran 90 never saw the light of

day. One vendor's extended (and much-copied) version of Fortran 77 had virtually become an industry standard, and it saw as its mission the maintenance of this lead. A new standard would cost it its perceived precious advantage. Other large vendors had similar points of view, although those marketing supercomputers were clearly keen on the array language. Most users, on the other hand, were hardly prepared to invest large amounts of their employers' and their own resources in simply settling for a trivial set of improvements to the existing standard. However, as long as X3J3 worked under a simple-majority voting rule, at least some apparent progress could be made, although the underlying differences often surfaced. These were even sometimes between users — those who wanted Fortran to become a truly modern language and those wanting to maintain indefinite backwards compatibility for their billions of lines of existing code.

At a watershed meeting, in Scranton, Pennsylvania, in 1986, held in an atmosphere that sometimes verged on despair, a fragile compromise was reached as a basis for further work. One breakthrough was to weaken the procedures for removing outdated features from the language, particularly by removing no features whatsoever from the next standard and by striking storage association (i.e., `COMMON` and `EQUIVALENCE`) from the list of features to be designated as obsolescent (as they are now known). A series of votes definitively removed from the language all plans to add: arrays of arrays, exception handling, nesting of internal procedures, the `FORALL` statement (now in Fortran 95), and a means to access skew array sections. There were other features on this list that, although removed, were reinstated at later meetings: user-defined operators, operator overloading, array and structure constructors, and vector-valued subscripts. After many more travails, the committee voted, a year later, by 26 votes to 9, to forward the document for what was to become the first of three periods of public comment.

While the document was going through the formal standards bureaucracy and being placed before the public, X3J3 polished it further. X3J3 also prepared procedures for processing the comments it anticipated receiving from the public, and to each of which, under the rules, it would have to reply individually. It was just as well. Roughly 400 replies flooded in, many of them very detailed and, disappointingly for those of us wanting a new standard quickly, unquestionably negative towards our work. For many it was too radical, but many others pleaded for yet more modern features, such as pointers.

Now the committee was deadlocked. Given that a document had already been published, any further change required not a simple but a two-thirds majority. The conservatives and the radicals could each block a move to modify the draft standard, or to accept a revised one for public review — and just that happened, in Champagne-Urbana, Illinois, in 1988. Any change, be it on the one hand to modify the list of obsolescent features, to add the pointers or bit data type wanted by the public, to add multi-byte characters to support Kanji and other non-European languages or, on the other hand, to emasculate the language by removing modules or operator overloading, and hence abstract data types, to name but some suggestions, none of these could be done individually or collectively in a way that would achieve consensus. I wrote:

"In my opinion, no standard can now emerge without either a huge concession by the users to the vendors (`MODULE` / `USE`) and/or a major change in the composition of the committee. I do not see how members who have worked for up to a decade

or more, devoting time and intellectual energy far beyond the call of duty, can be expected to make yet more personal sacrifices if no end to the work is in sight, or if that end is nothing but a travesty of what had been designed and intended as a modern scientific programming language. . . . I think the August meeting will be a watershed — if no progress is achieved there will be dramatic resignations, and ISO could even remove the work from ANSI, which is failing conspicuously in its task."

(However, the same notes began with a quotation from *The Taming of the Shrew*: "And do as adversaries do in law, / Strive mightily, but eat and drink / as friend." That we always did, copiously.)

Resolution

The "August meeting" was, unexpectedly, imbued with a spirit of compromise that had been so sadly lacking at the previous one. Nevertheless, after a week of discussing four separate plans to rescue the standard, no agreement was reached. Now the question seriously arose: Was X3J3 incapable of producing a new Fortran standard for the international community, doomed to eternal deadlock, a victim of ANSI procedures?

Breakthrough was achieved at a traumatic meeting of WG5 in Paris, France, a month later. The committee spent several extraordinary days drawing up a detailed list of what *it* wanted to be in Fortran 8x. Finally, it set X3J3 an ultimatum that was unprecedented in the standards world: The ANSI committee was to produce a new draft document, corresponding to WG5's wishes, within five months! Failing that, WG5 would assume responsibility and produce the new standard itself.

This decision was backed by the senior U.S. committee, X3, which effectively directed X3J3 to carry out WG5's wishes. And it did! The following November, it implemented most of the technical changes, adding pointers, bit manipulation intrinsic procedures, and vector-valued subscripts, and removing user-defined elemental functions (now in Fortran 95). The actual list of changes was much longer. X3J3 and WG5, now collaborating closely, often in gruelling six-day meetings, spent the next 18 months and two more periods of (positive) public comment putting the finishing touches to what was now called Fortran 90, and it was finally adopted, after some cliff-hanging votes, for forwarding as a U.S. and international standard on April 11, 1991, in Minneapolis, Minnesota.

Among the remaining issues that were decided along the way were whether pointers should be a data type or be defined in terms of an attribute of a variable, implying strong typing (the latter was chosen), whether the new standard should coexist alongside the old one rather than definitively replace it (it coexisted for a while in the U.S., but was a replacement elsewhere, under ISO rules), and whether, in the new free source form, blanks should be significant (fortunately, they are).

Fortran 90

The main new features of Fortran 90 are, first and foremost, the array language and abstract data types. The first is built on whole array operations and assignments, array sections, intrinsic procedures for arrays, and dynamic storage. It was designed with optimization in mind. The second is built on modules and module procedures, derived data types, operator overloading and generic interfaces, together with

pointers. Also important are the new facilities for numerical computation including a set of numeric inquiry functions, the parametrization of the intrinsic types, new control constructs — `SELECT CASE` and new forms of `DO`, internal and recursive procedures and optional and keyword arguments, improved I/O facilities, and many new intrinsic procedures. Last but not least are the new free source form, an improved style of attribute-oriented specifications, the `IMPLICIT NONE` statement, and a mechanism for identifying redundant features for subsequent removal from the language. The requirement on compilers to be able to identify, for example, syntax extensions, and to report why a program has been rejected, are also significant. The resulting language is not only a far more powerful tool than its successor, but a safer and more reliable one too. Storage association, with its attendant dangers, is not abolished, but rendered unnecessary. Indeed, experience shows that compilers detect errors far more frequently than before, resulting in a faster development cycle. The array syntax and recursion also allow quite compact code to be written, a further aid to safe programming.

No programming language can succeed if it consists simply of a definition (witness Algol 68). Also required are robust compilers from a wide variety of vendors, documentation at various levels, and a body of experience. The first Fortran 90 compiler appeared surprisingly quickly, in 1991, especially in view of the widely touted opinion that it would be very difficult to write one. Even more remarkable was that it was written by one person, Malcolm Cohen of NAG, in Oxford, U.K. There was a gap before other compilers appeared, but now they exist as native implementations for almost all leading computers, from the largest to PCs. For the most part, they produce very efficient object code; where, for certain new features, this is not the case, work is in progress to improve them.

The first book, *Fortran 90 Explained*, was published by John Reid and me shortly before the standard itself was published. Others followed in quick succession, including excellent texts aimed at the college market. At the time of writing there are at least 19 books in English and 22 in various other languages: Chinese, Dutch, French, Japanese, Russian, and Swedish. Thus, the documentation condition is fulfilled.

The body of experience, on the other hand, has yet to be built up to a critical size. Teaching of the language at college level has only just begun. However, I am certain that this present volume will contribute decisively to a significant breakthrough, as it provides models not only of the numerical algorithms for which previous editions are already famed, but also of an excellent Fortran 90 style, something that can develop only with time. Redundant features are abjured. It shows that, if we abandon these features and use new ones in their place, the appearance of code can initially seem unfamiliar, but, in fact, the advantages become rapidly apparent. This new edition of *Numerical Recipes* stands as a landmark in this regard.

Fortran Evolution

The formal procedures under which languages are standardized require them either to evolve or to die. A standard that has not been revised for some years must either be revised and approved anew, or be withdrawn. This matches the technical pressure on the language developers to accommodate the increasing complexity both of the problems to be tackled in scientific computation and of the underlying hardware

on which programs run. Increasing problem complexity requires more powerful features and syntax; new hardware needs language features that map onto it well.

Thus it was that X3J3 and WG5, having finished Fortran 90, began a new round of improvement. They decided very quickly on new procedures that would avoid the disputes that bedevilled the previous work: WG5 would decide on a plan for future standards, and X3J3 would act as the so-called development body that would actually produce them. This would be done to a strict timetable, such that any feature that could not be completed on time would have to wait for the next round. It was further decided that the next major revision should appear a decade after Fortran 90 but, given the somewhat discomforting number of requests for interpretation that had arrived, about 200, that a minor revision should be prepared for mid-term, in 1995. This should contain only "corrections, clarifications and interpretations" and a very limited number (some thought none) of minor improvements.

At the same time, scientific programmers were becoming increasingly concerned at the variety of methods that were necessary to gain efficient performance from the ever-more widely used parallel architectures. Each vendor provided a different set of parallel extensions for Fortran, and some academic researchers had developed yet others. On the initiative of Ken Kennedy of Rice University, a High-Performance Fortran Forum was established. A coalition of vendors and users, its aim was to produce an ad hoc set of extensions to Fortran that would become an informal but widely accepted standard for portable code. It set itself the daunting task of achieving that in just one year, and succeeded. Melding existing dialects like Fortran D, CM Fortran, and Vienna Fortran, and adopting the new Fortran 90 as a base, because of its array syntax, High-Performance Fortran (HPF) was published in 1993 and has since become widely implemented. However, although HPF was designed for data parallel codes and mainly implemented in the form of directives that appear to non-HPF processors as comment lines, an adequate functionality could not be achieved without extending the Fortran syntax. This was done in the form of the `PURE` attribute for functions — an assertion that they contain no side effects — and the `FORALL` construct — a form of array assignment expressed with the help of indices.

The dangers of having diverging or competing forms of Fortran 90 were immediately apparent, and the standards committees wisely decided to incorporate these two syntactic changes also into Fortran 95. But they didn't stop there. Two further extensions, useful not only for their expressive power but also to access parallel hardware, were added: elemental functions, ones written in terms of scalars but that accept array arguments of any permitted shape or size, and an extension to allow nesting of `WHERE` constructs, Fortran's form of masked assignment. To readers of *Numerical Recipes*, perhaps the most relevant of the minor improvements that Fortran 95 brings are the ability to distinguish between a negative and a positive real zero, automatic deallocation of allocatable arrays, and a means to initialize the values of components of objects of derived data types and to initialize pointers to null.

The medium-term objective of a relatively minor upgrade has been achieved on schedule. But what does the future hold? Developments in the underlying principles of procedural programming languages have not ceased. Early Fortran introduced the concepts of expression abstraction (`X=Y+Z`) and later control expression (e.g., the `DO` loop). Fortran 77 continued this with the `if-then-else`, and Fortran 90 with the `DO` and `SELECT CASE` constructs. Fortran 90 has a still higher level of expression abstraction (array assignments and expressions) as well as data structures and even

full-blown abstract data types. However, during the 1980s the concept of objects came to the fore, with methods bound to the objects on which they operate. Here, one particular language, C++, has come to dominate the field. Fortran 90 lacks a means to point to functions, but otherwise has most of the necessary features in place, and the standards committees are now faced with the dilemma of deciding whether to make the planned Fortran 2000 a fully object-oriented language. This could possibly jeopardize its powerful, and efficient, numerical capabilities by too great an increase in language complexity, so should they simply batten down the hatches and not defer to what might be only a passing storm? At the time of writing, this is an open issue. One issue that is not open is Fortran's lack of in-built exception handling. It is virtually certain that such a facility, much requested by the numerical community, and guided by John Reid, will be part of the next major revision. The list of other requirements is long but speculative, but some at the top of the list are conditional compilation, command line argument handling, I/O for objects of derived type, and asynchronous I/O (which is also planned for the next release of HPF). In the meantime, some particularly pressing needs have been identified, for the handling of floating-point exceptions, interoperability with C, and allowing allocatable arrays as structure components, dummy arguments, and function results. These have led WG5 to begin processing these three items using a special form of fast track, so that they might become optional but standard extensions well before Fortran 2000 itself is published in the year 2001.

Conclusion

Writing a book is always something of a gamble. Unlike a novel that stands or falls on its own, a book devoted to a programming language is dependent on the success of others, and so the risk is greater still. However, this new *Numerical Recipes in Fortran 90* volume is no ordinary book, since it comes as the continuation of a highly successful series, and so great is its significance that it can, in fact, influence the outcome in its own favor. I am entirely confident that its publication will be seen as an important event in the story of Fortran 90, and congratulate its authors on having performed a great service to the field of numerical computing.

Geneva, Switzerland Michael Metcalf
January 1996

License Information

Read this section if you want to use the programs in this book on a computer. You'll need to read the following Disclaimer of Warranty, get the programs onto your computer, and acquire a Numerical Recipes software license. (Without this license, which can be the free "immediate license" under terms described below, the book is intended as a text and reference book, for reading purposes only.)

Disclaimer of Warranty

We make no warranties, express or implied, that the programs contained in this volume are free of error, or are consistent with any particular standard of merchantability, or that they will meet your requirements for any particular application. They should not be relied on for solving a problem whose incorrect solution could result in injury to a person or loss of property. If you do use the programs in such a manner, it is at your own risk. The authors and publisher disclaim all liability for direct or consequential damages resulting from your use of the programs.

How to Get the Code onto Your Computer

Pick one of the following methods:

- You can type the programs from this book directly into your computer. In this case, the *only* kind of license available to you is the free "immediate license" (see below). You are not authorized to transfer or distribute a machine-readable copy to any other person, nor to have any other person type the programs into a computer on your behalf. We do not want to hear bug reports from you if you choose this option, because experience has shown that *virtually all* reported bugs in such cases are typing errors!
- You can download the Numerical Recipes programs electronically from the Numerical Recipes On-Line Software Store, located at our Web site (`http://www.nr.com`). They are packaged as a password-protected file, and you'll need to purchase a license to unpack them. You can get a single-screen license and password immediately, on-line, from the On-Line Store, with fees ranging from $50 (PC, Macintosh, educational institutions' UNIX) to $140 (general UNIX). Downloading the packaged software from the On-Line Store is also the way to start if you want to acquire a more general (multiscreen, site, or corporate) license.
- You can purchase media containing the programs from Cambridge University Press. Diskette versions are available in IBM-compatible format for machines running Windows 3.1, 95, or NT. CDROM versions in ISO-9660 format for PC, Macintosh, and UNIX systems are also available; these include both Fortran and C versions (as well as versions in Pascal

and BASIC from the first edition) on a single CDROM. Diskettes purchased from Cambridge University Press include a single-screen license for PC or Macintosh only. The CDROM is available with a single-screen license for PC or Macintosh (order ISBN 0 521 576083), or (at a slightly higher price) with a single-screen license for UNIX workstations (order ISBN 0 521 576075). Orders for media from Cambridge University Press can be placed at 800 872-7423 (North America only) or by email to orders@cup.org (North America) or trade@cup.cam.ac.uk (rest of world). Or, visit the Web sites `http://www.cup.org` (North America) or `http://www.cup.cam.ac.uk` (rest of world).

Types of License Offered

Here are the types of licenses that we offer. Note that some types are automatically acquired with the purchase of media from Cambridge University Press, or of an unlocking password from the Numerical Recipes On-Line Software Store, while other types of licenses require that you communicate specifically with Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739). Our Web site `http://www.nr.com` has additional information.

- ["Immediate License"] If you are the individual owner of a copy of this book and you type one or more of its routines into your computer, we authorize you to use them on that computer for your own personal and noncommercial purposes. You are not authorized to transfer or distribute machine-readable copies to any other person, or to use the routines on more than one machine, or to distribute executable programs containing our routines. This is the only free license.
- ["Single-Screen License"] This is the most common type of low-cost license, with terms governed by our Single Screen (Shrinkwrap) License document (complete terms available through our Web site). Basically, this license lets you use Numerical Recipes routines on any one screen (PC, workstation, X-terminal, etc.). You may also, under this license, transfer pre-compiled, executable programs incorporating our routines to other, unlicensed, screens or computers, providing that (i) your application is noncommercial (i.e., does not involve the selling of your program for a fee), (ii) the programs were first developed, compiled, and successfully run on a licensed screen, and (iii) our routines are bound into the programs in such a manner that they cannot be accessed as individual routines and cannot practicably be unbound and used in other programs. That is, under this license, your program user must not be able to use our programs as part of a program library or "mix-and-match" workbench. Conditions for other types of commercial or noncommercial distribution may be found on our Web site (`http://www.nr.com`).
- ["Multi-Screen, Server, Site, and Corporate Licenses"] The terms of the Single Screen License can be extended to designated groups of machines, defined by number of screens, number of machines, locations, or ownership. Significant discounts from the corresponding single-screen

prices are available when the estimated number of screens exceeds 40. Contact Numerical Recipes Software (email: orders@nr.com or fax: 781 863-1739) for details.

- ["Course Right-to-Copy License"] Instructors at accredited educational institutions who have adopted this book for a course, and who have already purchased a Single Screen License (either acquired with the purchase of media, or from the Numerical Recipes On-Line Software Store), may license the programs for use in that course as follows: Mail your name, title, and address; the course name, number, dates, and estimated enrollment; and advance payment of $5 per (estimated) student to Numerical Recipes Software, at this address: P.O. Box 243, Cambridge, MA 02238 (USA). You will receive by return mail a license authorizing you to make copies of the programs for use by your students, and/or to transfer the programs to a machine accessible to your students (but only for the duration of the course).

About Copyrights on Computer Programs

Like artistic or literary compositions, computer programs are protected by copyright. Generally it is an infringement for you to copy into your computer a program from a copyrighted source. (It is also not a friendly thing to do, since it deprives the program's author of compensation for his or her creative effort.) Under copyright law, all "derivative works" (modified versions, or translations into another computer language) also come under the same copyright as the original work.

Copyright does not protect ideas, but only the expression of those ideas in a particular form. In the case of a computer program, the ideas consist of the program's methodology and algorithm, including the necessary sequence of steps adopted by the programmer. The expression of those ideas is the program source code (particularly any arbitrary or stylistic choices embodied in it), its derived object code, and any other derivative works.

If you analyze the ideas contained in a program, and then express those ideas in your own completely different implementation, then that new program implementation belongs to you. That is what we have done for those programs in this book that are not entirely of our own devising. When programs in this book are said to be "based" on programs published in copyright sources, we mean that the ideas are the same. The expression of these ideas as source code is our own. We believe that no material in this book infringes on an existing copyright.

Trademarks

Several registered trademarks appear within the text of this book: Sun is a trademark of Sun Microsystems, Inc. SPARC and SPARCstation are trademarks of SPARC International, Inc. Microsoft, Windows 95, Windows NT, PowerStation, and MS are trademarks of Microsoft Corporation. DEC, VMS, Alpha AXP, and ULTRIX are trademarks of Digital Equipment Corporation. IBM is a trademark of International Business Machines Corporation. Apple and Macintosh are trademarks of Apple Computer, Inc. UNIX is a trademark licensed exclusively through X/Open

Co. Ltd. IMSL is a trademark of Visual Numerics, Inc. NAG refers to proprietary computer software of Numerical Algorithms Group (USA) Inc. PostScript and Adobe Illustrator are trademarks of Adobe Systems Incorporated. Last, and no doubt least, Numerical Recipes (when identifying products) is a trademark of Numerical Recipes Software.

Attributions

The fact that ideas are legally "free as air" in no way supersedes the ethical requirement that ideas be credited to their known originators. When programs in this book are based on known sources, whether copyrighted or in the public domain, published or "handed-down," we have attempted to give proper attribution. Unfortunately, the lineage of many programs in common circulation is often unclear. We would be grateful to readers for new or corrected information regarding attributions, which we will attempt to incorporate in subsequent printings.

Chapter 21. Introduction to Fortran 90 Language Features

21.0 Introduction

Fortran 90 is in many respects a backwards-compatible modernization of the long-used (and much abused) Fortran 77 language, but it is also, in other respects, a new language for parallel programming on present and future multiprocessor machines. These twin design goals of the language sometimes add confusion to the process of becoming fluent in Fortran 90 programming.

In a certain trivial sense, Fortran 90 is strictly backwards-compatible with Fortran 77. That is, any Fortran 90 compiler is supposed to be able to compile any legacy Fortran 77 code without error. The reason for terming this compatibility trivial, however, is that you have to tell the compiler (usually via a source file name ending in "`.f`" or "`.for`") that it is dealing with a Fortran 77 file. If you instead try to pass off Fortran 77 code as native Fortran 90 (e.g., by naming the source file something ending in "`.f90`") it will not always work correctly!

It is best, therefore, to approach Fortran 90 as a new computer language, albeit one with a lot in common with Fortran 77. Indeed, in such terms, Fortran 90 is a fairly *big* language, with a large number of new constructions and intrinsic functions. Here, in one short chapter, we do not pretend to provide a complete description of the language. Luckily, there are good books that do exactly that. Our favorite one is by Metcalf and Reid [1], cited throughout this chapter as "M&R." Other good starting points include [2] and [3].

Our goal, in the remainder of this chapter, is to give a good, working description of those Fortran 90 language features that are not immediately self-explanatory to Fortran 77 programmers, with particular emphasis on those that occur most frequently in the Fortran 90 versions of the Numerical Recipes routines. This chapter, by itself, will not teach you to write Fortran 90 code. But it ought to help you acquire a reading knowledge of the language, and perhaps provide enough of a head start that you can rapidly pick up the rest of what you need to know from M&R or another Fortran 90 reference book.

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press). [1]

Kerrigan, J.F. 1993, *Migrating to Fortran 90* (Sebastopol, CA: O'Reilly). [2]

Brainerd, W.S., Goldberg, C.H., and Adams, J.C. 1996, *Programmer's Guide to Fortran 90*, 3rd ed. (New York: Springer-Verlag). [3]

21.1 Quick Start: Using the Fortran 90 Numerical Recipes Routines

This section is for people who want to jump right in. We'll compute a Bessel function $J_0(x)$, where x is equal to the fourth root of the Julian Day number of the 200th full moon since January 1900. (Now *there's* a useful quantity!)

First, locate the important files `nrtype.f90`, `nrutil.f90`, and `nr.f90`, as listed in Appendices C1, C1, and C2, respectively. These contain *modules* that either are (i) used by our routines, or else (ii) describe the calling conventions of our routines to (your) user programs. Compile each of these files, producing (with most compilers) a `.mod` file and a `.o` (or similarly named) file for each one.

Second, create this main program file:

```
PROGRAM hello_bessel
USE nrtype
USE nr, ONLY: flmoon, bessj0
IMPLICIT NONE
INTEGER(I4B) :: n=200,nph=2,jd
REAL(SP) :: x,frac,ans
call flmoon(n,nph,jd,frac)
x=jd**0.25_sp
ans=bessj0(x)
write (*,*) 'Hello, Bessel: ', ans
END PROGRAM
```

Here is a quick explanation of some elements of the above program:

The first `USE` statement includes a module of ours named `nrtype`, whose purpose is to give symbolic names to some kinds of data types, among them single-precision reals ("`sp`") and four-byte integers ("`i4b`"). The second `USE` statement includes a module of ours that defines the calling sequences, and variable types, expected by (in this case) the Numerical Recipes routines `flmoon` and `bessj0`.

The `IMPLICIT NONE` statement signals that we want the compiler to require us explicitly to declare all variable types. *We strongly urge that you always take this option.*

The next two lines declare integer and real variables of the desired kinds. The variable `n` is initialized to the value 200, `nph` to 2 (a value expected by `flmoon`).

We call `flmoon`, and take the fourth root of the answer it returns as `jd`. Note that the constant 0.25 is typed to be single-precision by the appended `_sp`.

We call the `bessj0` routine, and print the answer.

Third, compile the main program file, and also the files `flmoon.f90`, `bessj0.f90`. Then, link the resulting object files with also `nrutil.o` (or similar system-dependent name, as produced in step 1). Some compilers will also require you to link with `nr.o` and `nrtype.o`.

Fourth, run the resulting executable file. Typical output is:

```
Hello, Bessel:   7.3096365E-02
```

21.2 Fortran 90 Language Concepts

The Fortran 90 language standard defines and uses a number of standard terms for concepts that occur in the language. Here we summarize briefly some of the most important concepts. Standard Fortran 90 terms are shown in *italics*. While by no means complete, the information in this section should help you get a quick start with your favorite Fortran 90 reference book or language manual.

A note on capitalization: Outside a character context, Fortran 90 is not case-sensitive, so you can use upper and lower case any way you want, to improve readability. A variable like `SP` (see below) is the same variable as the variable `sp`. We like to capitalize keywords whose use is primarily at compile-time (statements that delimit program and subprogram boundaries, declaration statements of variables, fixed parameter values), and use lower case for the bulk of run-time code. You can adopt any convention that you find helpful to your own programming style; but we strongly urge you to adopt and follow *some* convention.

Data Types and Kinds

Data types (also called simply *types*) can be either *intrinsic data types* (the familiar `INTEGER`, `REAL`, `LOGICAL`, and so forth) or else *derived data types* that are built up in the manner of what are called "structures" or "records" in other computer languages. (We'll use derived data types very sparingly in this book.) Intrinsic data types are further specified by their *kind parameter* (or simply *kind*), which is simply an integer. Thus, on many machines, `REAL(4)` (with `kind` $= 4$) is a single-precision real, while `REAL(8)` (with `kind` $= 8$) is a double-precision real. *Literal constants* (or simply *literals*) are specified as to kind by appending an underscore, as `1.5_4` for single precision, or `1.5_8` for double precision. [M&R, §2.5–§2.6]

Unfortunately, the specific integer values that define the different kind types are not specified by the language, but can vary from machine to machine. For that reason, one almost never uses literal kind parameters like 4 or 8, but rather defines in some central file, and imports into all one's programs, symbolic names for the kinds. For this book, that central file is the *module* named `nrtype`, and the chosen symbolic names include `SP`, `DP` (for reals); `I2B`, `I4B` (for two- and four-byte integers); and `LGT` for the default logical type. You will therefore see us consistently writing `REAL(SP)`, or `1.5_sp`, and so forth.

Here is an example of declaring some variables, including a one-dimensional array of length 500, and a two-dimensional array with 100 rows and 200 columns:

```
USE nrtype
REAL(SP) :: x,y,z
INTEGER(I4B) :: i,j,k
REAL(SP), DIMENSION(500) :: arr
REAL(SP), DIMENSION(100,200) :: barr
REAL(SP) :: carr(500)
```

The last line shows an alternative form for array syntax. And yes, there *are* default kind parameters for each intrinsic type, but these vary from machine to machine and can get you into trouble when you try to move code. We therefore specify all kind parameters explicitly in almost all situations.

Array Shapes and Sizes

The *shape* of an *array* refers to both its dimensionality (called its *rank*), and also the lengths along each dimension (called the *extents*). The shape of an array is specified by a rank-one array whose elements are the extents along each dimension, and can be queried with the `shape` intrinsic (see p. 949). Thus, in the above example, `shape(barr)` returns an array of length 2 containing the values $(100, 200)$.

The *size* of an array is its total number of elements, so the intrinsic `size(barr)` would return 20000 in the above example. More often one wants to know the extents along each dimension, separately: `size(barr,1)` returns the value 100, while `size(barr,2)` returns the value 200. [M&R, §2.10]

Section §21.3, below, discusses additional aspects of arrays in Fortran 90.

Memory Management

Fortran 90 is greatly superior to Fortran 77 in its memory-management capabilities, seen by the user as the ability to create, expand, or contract workspace for programs. Within *subprograms* (that is, *subroutines* and *functions*), one can have *automatic arrays* (or other *automatic data objects*) that come into existence each time the subprogram is entered, and disappear (returning their memory to the pool) when the subprogram is exited. The size of automatic objects can be specified by arbitrary expressions involving values passed as *actual arguments* in the calling program, and thus received by the subprogram through its corresponding *dummy arguments*. [M&R, §6.4]

Here is an example that creates some automatic workspace named `carr`:

```
SUBROUTINE dosomething(j,k)
USE nrtype
REAL(SP), DIMENSION(2*j,k**2) :: carr
```

Finer control on when workspace is created or destroyed can be achieved by declaring *allocatable arrays*, which exist as names only, without associated memory, until they are *allocated* within the program or subprogram. When no longer needed, they can be *deallocated*. The *allocation status* of an allocatable array can be tested by the program via the `allocated` intrinsic function (p. 952). [M&R, §6.5]

Here is an example in outline:

```
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: darr
...
allocate(darr(10,20))
...
deallocate(darr)
...
allocate(darr(100,200))
...
deallocate(darr)
```

Notice that `darr` is originally declared with only "slots" (colons) for its dimensions, and is then allocated/deallocated twice, with different sizes.

Yet finer control is achieved by the use of *pointers*. Like an allocatable array, a pointer can be allocated, at will, its own associated memory. However, it has the additional flexibility of alternatively being *pointer associated* with a *target* that

already exists under another name. Thus, pointers can be used as redefinable aliases for other variables, arrays, or (see §21.3) *array sections*. [M&R, §6.12]

Here is an example that first associates the pointer `parr` with the array `earr`, then later cancels that association and allocates it its own storage of size 50:

```
REAL(SP), DIMENSION(:), POINTER :: parr
REAL(SP), DIMENSION(100), TARGET :: earr
...
parr => earr
...
nullify(parr)
allocate(parr(50))
...
deallocate(parr)
```

Procedure Interfaces

When a procedure is *referenced* (e.g., called) from within a program or subprogram (examples of *scoping units*), the scoping unit must be told, or must deduce, the procedure's *interface*, that is, its calling sequence, including the types and kinds of all dummy arguments, returned values, etc. The recommended procedure is to specify this interface via an *explicit interface*, usually an *interface block* (essentially a declaration statement for subprograms) in the calling subprogram or in some *module* that the calling program includes via a `USE` statement. In this book all interfaces are explicit, and the module named `nr` contains interface blocks for all of the Numerical Recipes routines. [M&R, §5.11]

Here is a typical example of an interface block:

```
INTERFACE
   SUBROUTINE caldat(julian,mm,id,iyyy)
   USE nrtype
   INTEGER(I4B), INTENT(IN) :: julian
   INTEGER(I4B), INTENT(OUT) :: mm,id,iyyy
   END SUBROUTINE caldat
END INTERFACE
```

Once this interface is made known to a program that you are writing (by either explicit inclusion or a `USE` statement), then the compiler is able to flag for you a variety of otherwise difficult-to-find bugs. Although interface blocks can sometimes seem overly wordy, they give a big payoff in ultimately minimizing programmer time and frustration.

For compatibility with Fortran 77, the language also allows for *implicit interfaces*, where the calling program tries to figure out the interface by the old rules of Fortran 77. These rules are quite limited, and prone to producing devilishly obscure program bugs. We strongly recommend that implicit interfaces never be used.

Elemental Procedures and Generic Interfaces

Many *intrinsic procedures* (those defined by the language standard and thus usable without any further definition or specification) are also *generic*. This means that a single procedure name, such as `log(x)`, can be used with a variety of types and kind parameters for the argument `x`, and the result returned will have the same type and kind parameter as the argument. In this example, `log(x)` allows any real or complex argument type.

Better yet, most generic functions are also *elemental*. The argument of an elemental function can be an array of arbitrary shape! Then, the returned result is an array of the same shape, with each element containing the result of applying the function to the corresponding element of the argument. (Hence the name *elemental*, meaning "applied element by element.") [M&R, §8.1] For example:

```
REAL(SP), DIMENSION(100,100) :: a,b
b=sin(a)
```

Fortran 90 has no facility for creating new, user-defined elemental functions. It does have, however, the related facility of *overloading* by the use of *generic interfaces*. This is invoked by the use of an interface block that attaches a single *generic name* to a number of distinct subprograms whose dummy arguments have different types or kinds. Then, when the generic name is referenced (e.g., called), the compiler chooses the specific subprogram that matches the types and kinds of the actual arguments used. [M&R, §5.18] Here is an example of a generic interface block:

```
INTERFACE myfunc
   FUNCTION myfunc_single(x)
   USE nrtype
   REAL(SP) :: x,myfunc_single
   END FUNCTION myfunc_single

   FUNCTION myfunc_double(x)
   USE nrtype
   REAL(DP) :: x,myfunc_double
   END FUNCTION myfunc_double
END INTERFACE
```

A program with knowledge of this interface could then freely use the function reference `myfunc(x)` for `x`'s of both type `SP` and type `DP`.

We use overloading quite extensively in this book. A typical use is to provide, under the same name, both scalar and vector versions of a function such as a Bessel function, or to provide both single-precision and double-precision versions of procedures (as in the above example). Then, to the extent that we have provided all the versions that you need, you can pretend that our routine is elemental. In such a situation, if you ever call our function with a type or kind that we have *not* provided, the compiler will instantly flag the problem, because it is unable to resolve the generic interface.

Modules

Modules, already referred to several times above, are Fortran 90's generalization of Fortran 77's common blocks, `INCLUDE`d files of parameter statements, and (to some extent) statement functions. Modules are *program units*, like main programs or subprograms (subroutines and functions), that can be separately compiled. A module is a convenient place to stash global data, *named constants* (what in Fortran 77 are called "symbolic constants" or "`PARAMETER`s"), interface blocks to subprograms and/or actual subprograms themselves (*module subprograms*). The convenience is that a module's information can be incorporated into another program unit via a simple, one-line `USE` statement. [M&R, §5.5]

Here is an example of a simple module that defines a few parameters, creates some global storage for an array named `arr` (as might be done with a Fortran 77 common block), and defines the interface to a function `yourfunc`:

```
MODULE mymodule
   USE nrtype
   REAL(SP), PARAMETER :: con1=7.0_sp/3.0_sp,con2=10.0_sp
   INTEGER(I4B), PARAMETER :: ndim=10,mdim=9
   REAL(SP), DIMENSION(ndim,mdim) :: arr
   INTERFACE
      FUNCTION yourfunc(x)
      USE nrtype
      REAL(SP) :: x,yourfunc
      END FUNCTION yourfunc
   END INTERFACE
END MODULE mymodule
```

As mentioned earlier, the module `nr` contains `INTERFACE` declarations for all the Numerical Recipes. When we include a statement of the form

```
USE nr, ONLY: recipe1
```

it means that the program uses the additional routine `recipe1`. The compiler is able to use the explicit interface declaration in the module to check that `recipe1` is invoked with arguments of the correct type, shape, and number. However, a weakness of Fortran 90 is that there is no fail-safe way to be sure that the interface module (here `nr`) stays synchronized with the underlying routine (here `recipe1`). You might think that you could accomplish this by putting `USE nr, ONLY: recipe1` into the `recipe1` program itself. Unfortunately, the compiler interprets this as an erroneous double definition of `recipe1`'s interface, rather than (as would be desirable) as an opportunity for a consistency check. (To achieve this kind of consistency check, you can put the procedures themselves, not just their interfaces, into the module.)

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.3 More on Arrays and Array Sections

Arrays are the central conceptual core of Fortran 90 as a *parallel* programming language, and thus worthy of some further discussion. We have already seen that arrays can "come into existence" in Fortran 90 in several ways, either directly declared, as

```
REAL(SP), DIMENSION(100,200) :: arr
```

or else allocated by an *allocatable* variable or a *pointer* variable,

```
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: arr
REAL(SP), DIMENSION(:,:), POINTER :: barr
...
allocate(arr(100,200),barr(100,200))
```

or else (not previously mentioned) passed into a subprogram through a dummy argument:

```
SUBROUTINE myroutine(carr)
USE nrtype
REAL(SP), DIMENSION(:,:) :: carr
...
i=size(carr,1)
```

```
j=size(carr,2)
```

In the above example we also show how the subprogram can find out the size of the actual array that is passed, using the `size` intrinsic. This routine is an example of the use of an *assumed-shape array*, new to Fortran 90. The actual extents along each dimension are inherited from the calling routine at run time. A subroutine with assumed-shape array arguments *must* have an explicit interface in the calling routine, otherwise the compiler doesn't know about the extra information that must be passed. A typical setup for calling `myroutine` would be:

```
PROGRAM use_myroutine
USE nrtype
REAL(SP), DIMENSION(10,10) :: arr
INTERFACE
   SUBROUTINE myroutine(carr)
   USE nrtype
   REAL(SP), DIMENSION(:,:) :: carr
   END SUBROUTINE myroutine
END INTERFACE
...
call myroutine(a)
```

Of course, for the recipes we have provided all the interface blocks in the file `nr.f90`, and you need only a `USE nr` statement in your calling program.

Conformable Arrays

Two arrays are said to be *conformable* if their shapes are the same. Fortran 90 allows practically all operations among conformable arrays and elemental functions that are allowed for scalar variables. Thus, if `arr`, `barr`, and `carr` are mutually conformable, we can write,

```
arr=barr+cos(carr)+2.0_sp
```

and have the indicated operations performed, element by corresponding element, on the entire arrays. The above line also illustrates that a scalar (here the constant `2.0_sp`, but a scalar variable would also be fine) is deemed conformable with *any* array — it gets "expanded" to the shape of the rest of the expression that it is in. [M&R, §3.11]

In Fortran 90, as in Fortran 77, the default lower bound for an array subscript is 1; however, it can be made some other value at the time that the array is declared:

```
REAL(SP), DIMENSION(100,200) :: farr
REAL(SP), DIMENSION(0:99,0:199) :: garr
...
farr = 3.0_sp*garr + 1.0_sp
```

Notice that `farr` and `garr` are conformable, since they have the same shape, in this case $(100, 200)$. Also note that when they are used in an array expression, the operations are taken between the corresponding elements *of their shapes*, not necessarily the corresponding elements *of their indices*. [M&R, §3.10] In other words, one of the components evaluated is,

```
farr(1,1) = 3.0_sp*garr(0,0) + 1.0_sp
```

This illustrates a fundamental aspect of array (or data) parallelism in Fortran 90. Array constructions should *not* be thought of as merely abbreviations for do-loops

over indices, but rather as genuinely parallel operations on same-shaped objects, abstracted of their indices. This is why the standard makes no statement about the order in which the individual operations in an array expression are executed; they might in fact be carried out simultaneously, on parallel hardware.

By default, array expressions and assignments are performed for all elements of the same-shaped arrays referenced. This can be modified, however, by use of a `where` construction like this:

```
where (harr > 0.0_sp)
   farr = 3.0_sp*garr + 1.0_sp
end where
```

Here `harr` must also be conformable to `farr` and `garr`. Analogously with the Fortran `if`-statement, there is also a one-line form of the `where`-statement. There is also a `where ... elsewhere ... end where` form of the statement, analogous to `if ... else if ... end if`. A significant language limitation in Fortran 90 is that nested `where`-statements are not allowed. [M&R, §6.8]

Array Sections

Much of the versatility of Fortran 90's array facilities stems from the availability of *array sections*. An array section acts like an array, but its memory location, and thus the values of its elements, is actually a subset of the memory location of an already-declared array. Array sections are thus "windows into arrays," and they can appear on either the left side, or the right side, or both, of a replacement statement. Some examples will clarify these ideas.

Let us presume the declarations

```
REAL(SP), DIMENSION(100) :: arr
INTEGER(I4B), DIMENSION(6) :: iarr=(/11,22,33,44,55,66/)
```

Note that `iarr` is not only declared, it is also initialized by an *initialization expression* (a replacement for Fortran 77's `DATA` statement). [M&R, §7.5] Here are some array sections constructed from these arrays:

Array Section	*What It Means*
`arr(:)`	same as `arr`
`arr(1:100)`	same as `arr`
`arr(1:10)`	one-dimensional array containing first 10 elements of `arr`
`arr(51:100)`	one-dimensional array containing second half of `arr`
`arr(51:)`	same as `arr(51:100)`
`arr(10:1:-1)`	one-dimensional array containing first 10 elements of `arr`, but in *reverse order*
`arr( (/10,99,1,6/) )`	one-dimensional array containing elements 10, 99, 1, and 6 of `arr`, in that order
`arr(iarr)`	one-dimensional array containing elements 11, 22, 33, 44, 55, 66 of `arr`, in that order

Now let's try some array sections of the two-dimensional array

```
REAL(SP), DIMENSION(100,100) :: barr
```

Array Section	*What It Means*
`barr(:,:)`	same as `barr`
`barr(1:100,1:100)`	same as `barr`
`barr(7,:)`	one-dimensional array containing the 7th row of `barr`
`barr(7,1:100)`	same as `barr(7,:)`
`barr(:,7)`	one-dimensional array containing the 7th column of `barr`
`barr(21:30,71:90)`	two-dimensional array containing the sub-block of `barr` with the indicated ranges of indices; the shape of this array section is $(10, 20)$
`barr(100:1:-1,100:1:-1)`	two-dimensional array formed by flipping `barr` upside down and backwards
`barr(2:100:2,2:100:2)`	two-dimensional array of shape $(50, 50)$ containing the elements of `barr` whose row and column indices are both even

Some terminology: A construction like `2:100:2`, above, is called a *subscript triplet*. Its integer pieces (which may be integer constants, or more general integer expressions) are called *lower*, *upper*, and *stride*. Any of the three may be omitted. An omitted stride defaults to the value 1. Notice that, if $(\textit{upper} - \textit{lower})$ has a different sign from *stride*, then a subscript triplet defines an empty or *zero-length* array, e.g., `1:5:-1` or `10:1:1` (or its equivalent form, simply `10:1`). Zero-length arrays are not treated as errors in Fortran 90, but rather as "no-ops." That is, no operation is performed in an expression or replacement statement among zero-length arrays. (This is essentially the same convention as in Fortran 77 for do-loop indices, which array expressions often replace.) [M&R, §6.10]

It is important to understand that array sections, when used in array expressions, match elements with other parts of the expression *according to shape*, not according to indices. (This is exactly the same principle that we applied, above, to arrays with subscript lower bounds different from the default value of 1.) One frequently exploits this feature in using array sections to carry out operations on arrays that access neighboring elements. For example,

```
carr(1:n-1,1:n-1) = barr(1:n-1,1:n-1)+barr(2:n,2:n)
```

constructs in the $(n-1) \times (n-1)$ matrix `carr` the sum of each of the corresponding elements in $n \times n$ `barr` added to its diagonally lower-right neighbor.

Pointers are often used as aliases for array sections, especially if the same array sections are used repeatedly. [M&R, §6.12] For example, with the setup

```
REAL(SP), DIMENSION(:,:), POINTER :: leftb,rightb
```

```
leftb=>barr(1:n-1,1:n-1)
rightb=>barr(2:n,2:n)
```

the statement above can be coded as

```
carr(1:n-1,1:n-1)=leftb+rightb
```

We should also mention that array sections, while powerful and concise, are sometimes not quite powerful enough. While any row or column of a matrix is easily accessible as an array section, there is no good way, in Fortran 90, to access (e.g.) the diagonal of a matrix, even though its elements are related by a linear progression in the Fortran storage order (by columns). These so-called *skew-sections* were much discussed by the Fortran 90 standards committee, but they were not implemented. We will see examples later in this volume of work-around programming tricks (none totally satisfactory) for this omission. (Fortran 95 corrects the omission; see §21.6.)

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.4 Fortran 90 Intrinsic Procedures

Much of Fortran 90's power, both for parallel programming and for its concise expression of algorithmic ideas, comes from its rich set of intrinsic procedures. These have the effect of making the language "large," hence harder to learn. However, effort spent on learning to use the intrinsics — particularly some of their more obscure, and more powerful, optional arguments — is often handsomely repaid.

This section summarizes the intrinsics that we find useful in numerical work. We omit, here, discussion of intrinsics whose exclusive use is for character and string manipulation. We intend only a summary, not a complete specification, which can be found in M&R's Chapter 8, or other reference books.

If you find the sheer number of new intrinsic procedures daunting, you might want to start with our list of the "top 10" (with the number of different Numerical Recipes routines that use each shown in parentheses): `size` (254), `sum` (44), `dot_product` (31), `merge` (27), `all` (25), `maxval` (23), `matmul` (19), `pack` (18), `any` (17), and `spread` (15). (Later, in Chapter 23, you can compare these numbers with our frequency of using the short utility functions that we define in a module named `nrutil` — several of which we think ought to have been included as Fortran 90 intrinsic procedures.)

The type, kind, and shape of the value returned by intrinsic functions will usually be clear from the short description that we give. As an additional hint (though not necessarily a precise description), we adopt the following codes:

Hint	*What It Means*
[Int]	an `INTEGER` kind type
[Real]	a `REAL` kind type
[Cmplx]	a `COMPLEX` kind type
[Num]	a numerical type and kind
[Lgcl]	a `LOGICAL` kind type
[Iarr]	a one-dimensional `INTEGER` array
[argTS]	same type and shape as the first argument
[argT]	same type as the first argument, but not necessarily the same shape

Numerical Elemental Functions

Little needs to be said about the numerical functions with identical counterparts in Fortran 77: `abs`, `acos`, `aimag`, `asin`, `atan`, `atan2`, `conjg`, `cos`, `cosh`, `dim`, `exp`, `log`, `log10`, `max`, `min`, `mod`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`. In Fortran 90 these are all *elemental* functions, so that any plausible type, kind, and shape of argument may be used. Except for `aimag`, which returns a real type from a complex argument, these all return [argTS] (see table above).

Although Fortran 90 recognizes, for compatibility, Fortran 77's so-called *specific names* for these functions (e.g., `iabs`, `dabs`, and `cabs` for the generic `abs`), these are entirely superfluous and should be avoided.

Fortran 90 corrects some ambiguity (or at least inconvenience) in Fortran 77's `mod(a,p)` function, by introducing a new function `modulo(a,p)`. The functions are essentially identical for positive arguments, but for negative `a` and positive `p`, `modulo` gives results more compatible with one's mathematical expectation that the answer should always be in the positive range 0 to `p`. E.g., `modulo(11,5)=1`, and `modulo(-11,5)=4`. [M&R, §8.3.2]

Conversion and Truncation Elemental Functions

Fortran 90's conversion (or, in the language of C, casting) and truncation functions are generally modeled on their Fortran 77 antecedents, but with the addition of an optional second integer argument, `kind`, that determines the kind of the result. Note that, if kind is omitted, you get a default kind — not necessarily related to the kind of your argument. The kind of the argument is of course known to the compiler by its previous declaration. Functions in this category (see below for explanation of arguments in slanted type) are:

[Real] `aint(a,`*`kind`*`)`
: Truncate to integer value, return as a real kind.

[Real] `anint(a,`*`kind`*`)`
: Nearest whole number, return as a real kind.

[Cmplx] `cmplx(x,`*`y,kind`*`)`

Convert to complex kind. If y is omitted, it is taken to be 0.

[Int] `int(a,`*`kind`*`)`
Convert to integer kind, truncating towards zero.

[Int] `nint(a,`*`kind`*`)`
Convert to integer kind, choosing the nearest whole number.

[Real] `real(a,`*`kind`*`)`
Convert to real kind.

[Lgcl] `logical(a,`*`kind`*`)`
Convert one logical kind to another.

We must digress here to explain the use of *optional arguments* and *keywords* as Fortran 90 language features. [M&R, §5.13] When a routine (either intrinsic or user-defined) has arguments that are declared to be optional, then the dummy names given to them also become keywords that distinguish — independent of their position in a calling list — which argument is intended to be passed. (There are some additional rules about this that we will not try to summarize here.) In this section's tabular listings, we indicate optional arguments in intrinsic routines by printing them in smaller slanted type. For example, the intrinsic function

`eoshift(array,shift,`*`boundary,dim`*`)`

has two required arguments, `array` and `shift`, and two optional arguments, `boundary` and `dim`. Suppose we want to call this routine with the actual arguments `myarray`, `myshift`, and `mydim`, but omitting the argument in the `boundary` slot. We do this by the expression

```
eoshift(myarray,myshift,dim=mydim)
```

Conversely, if we wanted a `boundary` argument, but no `dim`, we might write

```
eoshift(myarray,myshift,boundary=myboundary)
```

It is always a good idea to use this kind of keyword construction when invoking optional arguments, even though the rules allow keywords to be omitted in some unambiguous cases. Now back to the lists of intrinsic routines.

A peculiarity of the `real` function derives from its use both as a type conversion and for extracting the real part of complex numbers (related, but not identical, usages): If the argument of `real` is complex, and `kind` is omitted, then the result *isn't* a default real kind, but rather *is* (as one generally would want) the `real` kind type corresponding to the kind type of the complex argument, that is, single-precision real for single-precision complex, double-precision for double-precision, and so on. [M&R, §8.3.1] We recommend *never* using `kind` when you intend to extract the real part of a complex, and *always* using `kind` when you intend conversion of a real or integer value to a particular kind of `REAL`. (Use of the deprecated function `dble` is not recommended.)

The last two conversion functions are the exception in that they *don't* allow a `kind` argument, but rather return default integer kinds. (The X3J3 standards committee has fixed this in Fortran 95.)

[Int] `ceiling(a)`
Convert to integer, truncating towards more positive.

[Int] `floor(a)`
Convert to integer, truncating towards more negative.

Reduction and Inquiry Functions on Arrays

These are mostly the so-called *transformational functions* that accept array arguments and return either scalar values or else arrays of lesser rank. [M&R, §8.11] With no optional arguments, such functions act on all the elements of their single array argument, regardless of its shape, and produce a scalar result. When the optional argument `dim` is specified, they instead act on all one-dimensional sections that span the dimension `dim`, producing an answer one rank lower than the first argument (that is, omitting the `dim` dimension from its shape). When the optional argument `mask` is specified, only the elements with a corresponding true value in `mask` are scanned.

[Lgcl] `all(mask,dim)`
Returns true if all elements of `mask` are true, false otherwise.

[Lgcl] `any(mask,dim)`
Returns true if any of the elements of `mask` are true, false otherwise.

[Int] `count(mask,dim)`
Counts the true elements in `mask`.

[Num] `maxval(array,dim,mask)`
Maximum value of the array elements.

[Num] `minval(array,dim,mask)`
Minimum value of the array elements.

[Num] `product(array,dim,mask)`
Product of the array elements.

[Int] `size(array,dim)`
Size (total number of elements) of `array`, or its extent along dimension `dim`.

[Num] `sum(array,dim,mask)`
Sum of the array elements.

The use of the `dim` argument can be confusing, so an example may be helpful. Suppose we have

$$\texttt{myarray} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

where, as always, the `i` index in `array(i,j)` numbers the rows while the `j` index numbers the columns. Then

$$\texttt{sum(myarray,dim=1)} = (15, 18, 21, 24)$$

that is, the `i` indices are "summed away" leaving only a `j` index on the result; while

$$\texttt{sum(myarray,dim=2)} = (10, 26, 42)$$

that is, the `j` indices are "summed away" leaving only an `i` index on the result. Of course we also have

$$\texttt{sum(myarray)} = 78$$

Two related functions return the location of particular elements in an array. The returned value is a one-dimensional integer array containing the respective subscript of the element along each dimension. Note that when the argument object is a *one*-dimensional array, the returned object is an integer *array of length 1*, not simply an integer. (Fortran 90 distinguishes between these.)

[Iarr] `maxloc(array,`*`mask`*`)`
Location of the maximum value in an array.

[Iarr] `minloc(array,`*`mask`*`)`
Location of the minimum value in an array.

Similarly returning a one-dimensional integer array are

[Iarr] `shape(array)`
Returns the shape of `array` as a one-dimensional integer array.

[Iarr] `lbound(array,`*`dim`*`)`
When `dim` is absent, returns an array of lower bounds for each dimension of subscripts of `array`. When `dim` is present, returns the value only for dimension `dim`, as a scalar.

[Iarr] `ubound(array,`*`dim`*`)`
When `dim` is absent, returns an array of upper bounds for each dimension of subscripts of `array`. When `dim` is present, returns the value only for dimension `dim`, as a scalar.

Array Unary and Binary Functions

The most powerful array operations are simply built into the language as operators. All the usual arithmetic and logical operators (`+`, `-`, `*`, `/`, `**`, `.not.`, `.and.`, `.or.`, `.eqv.`, `.neqv.`) can be applied to arrays of arbitrary shape or (for the binary operators) between two arrays of the same shape, or between arrays and scalars. The types of the arrays must, of course, be appropriate to the operator used. The result in all cases is to perform the operation element by element on the arrays.

We also have the intrinsic functions,

[Num] `dot_product(veca,vecb)`
Scalar dot product of two one-dimensional vectors `veca` and `vecb`.

[Num] `matmul(mata,matb)`
Result of matrix-multiplying the two two-dimensional matrices `mata` and `matb`. The shapes have to be such as to allow matrix multiplication. Vectors (one-dimensional arrays) are additionally allowed as either the first or second argument, but not both; they are treated as row vectors in the first argument, and as column vectors in the second.

You might wonder how to form the *outer* product of two vectors, since `matmul` specifically excludes this case. (See §22.1 and §23.5 for answer.)

Array Manipulation Functions

These include many powerful features that a good Fortran 90 programmer should master.

[argTS] `cshift(array,shift,`*`dim`*`)`
: If `dim` is omitted, it is taken to be 1. Returns the result of circularly left-shifting every one-dimensional section of `array` (in dimension `dim`) by `shift` (which may be negative). That is, for positive `shift`, values are moved to smaller subscript positions. Consult a Fortran 90 reference (e.g., [M&R, §8.13.5]) for the case where `shift` is an array.

[argTS] `merge(tsource,fsource,mask)`
: Returns same shape object as `tsource` and `fsource` containing the former's components where `mask` is true, the latter's where it is false.

[argTS] `eoshift(array,shift,`*`boundary,dim`*`)`
: If `dim` is omitted, it is taken to be 1. Returns the result of end-off left-shifting every one-dimensional section of `array` (in dimension `dim`) by `shift` (which may be negative). That is, for positive `shift`, values are moved to smaller subscript positions. If `boundary` is present as a scalar, it supplies elements to fill in the blanks; if it is not present, zero values are used. Consult a Fortran 90 reference (e.g., [M&R, §8.13.5]) for the case where `boundary` and/or `shift` is an array.

[argT] `pack(array,mask,`*`vector`*`)`
: Returns a one-dimensional array containing the elements of `array` that pass the `mask`. Components of optional `vector` are used to pad out the result to the size of `vector` with specified values.

[argT] `reshape(source,shape,`*`pad,order`*`)`
: Takes the elements of `source`, in normal Fortran order, and returns them (as many as will fit) as an array whose shape is specified by the one-dimensional integer array `shape`. If there is space remaining, then `pad` must be specified, and is used (as many sequential copies as necessary) to fill out the rest. For description of `order`, consult a Fortran 90 reference, e.g., [M&R, 8.13.3].

[argT] `spread(source,dim,ncopies)`
: Returns an array whose rank is one greater than `source`, and whose `dim` dimension is of length `ncopies`. Each of the result's `ncopies` array sections having a fixed subscript in dimension `dim` is a copy of `source`. (That is, it spreads `source` into the `dim`th dimension.)

[argT] `transpose(matrix)`
: Returns the transpose of `matrix`, which must be two-dimensional.

[argT] `unpack(vector,mask,field)`
: Returns an array whose type is that of `vector`, but whose shape is that of `mask`. The components of `vector` are put, in order, into the positions where `mask` is true. Where `mask` is false, components of `field` (which may be a scalar or an array with the same shape as `mask`) are used instead.

Bitwise Functions

Most of the bitwise functions should be familiar to Fortran 77 programmers as longstanding standard extensions of that language. Note that the bit *positions* number from zero to one less than the value returned by the `bit_size` function. Also note that bit positions number *from right to left*. Except for `bit_size`, the following functions are all elemental.

[Int] `bit_size(i)`
Number of bits in the integer type of `i`.

[Lgcl] `btest(i,pos)`
True if bit position `pos` is 1, false otherwise.

[Int] `iand(i,j)`
Bitwise logical and.

[Int] `ibclr(i,pos)`
Returns `i` but with bit position `pos` set to zero.

[Int] `ibits(i,pos,len)`
Extracts `len` consecutive bits starting at position `pos` and puts them in the low bit positions of the returned value. (The high positions are zero.)

[Int] `ibset(i,pos)`
Returns `i` but with bit position `pos` set to 1.

[Int] `ieor(i,j)`
Bitwise exclusive or.

[Int] `ior(i,j)`
Bitwise logical or.

[Int] `ishft(i,shift)`
Bitwise left shift by `shift` (which may be negative) with zeros shifted in from the other end.

[Int] `ishftc(i,shift)`
Bitwise circularly left shift by `shift` (which may be negative).

[Int] `not(i)`
Bitwise logical complement.

Some Functions Relating to Numerical Representations

[Real] `epsilon(x)`
Smallest nonnegligible quantity relative to 1 in the numerical model of `x`.

[Num] `huge(x)`
Largest representable number in the numerical model of `x`.

[Int] `kind(x)`

Returns the kind value for the numerical model of `x`.

[Real] `nearest(x,s)`
Real number nearest to `x` in the direction specified by the sign of `s`.

[Real] `tiny(x)`
Smallest positive number in the numerical model of `x`.

Other Intrinsic Procedures

[Lgcl] `present(a)`
True, within a subprogram, if an optional argument is actually present, otherwise false.

[Lgcl] `associated(pointer,target)`
True if `pointer` is associated with `target` or (if `target` is absent) with any target, otherwise false.

[Lgcl] `allocated(array)`
True if the allocatable `array` is allocated, otherwise false.

There are some pitfalls in using `associated` and `allocated`, having to do with arrays and pointers that can find themselves in *undefined* status [see §21.5, and also M&R, §3.3 and §6.5.1]. For example, pointers are always "born" in an undefined status, where the `associated` function returns unpredictable values.

For completeness, here is a list of Fortran 90's intrinsic procedures not already mentioned:

Other Numerical Representation Functions: `digits`, `exponent`, `fraction`, `rrspacing`, `scale`, `set_exponent`, `spacing`, `maxexponent`, `minexponent`, `precision`, `radix`, `range`, `selected_int_kind`, `selected_real_kind`.

Lexical comparison: `lge`, `lgt`, `lle`, `llt`.

Character functions: `ichar`, `char`, `achar`, `iachar`, `index`, `adjustl`, `adjustr`, `len_trim`, `repeat`, `scan`, `trim`, `verify`.

Other: `mvbits`, `transfer`, `date_and_time`, `system_clock`, `random_seed`, `random_number`. (We will discuss random numbers in some detail in Chapter B7.)

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.5 Advanced Fortran 90 Topics

Pointers, Arrays, and Memory Management

One of the biggest improvements in Fortran 90 over Fortran 77 is in the handling of arrays, which are the cornerstone of many numerical algorithms. In this subsection we will take a closer look at how to use some of these new array features effectively. We will look at how to code certain commonly occurring elements of program design, and we will pay particular attention to avoiding "memory leaks," where — usually inadvertently — we keep cumulatively allocating new storage for an array, every time some piece of code is invoked.

Let's first review some of the rules for using allocatable arrays and pointers to arrays. Recall that a pointer is born with an undefined status. Its status changes to "associated" when you make it refer to a target, and to "disassociated" when you `nullify` the pointer. [M&R, §3.3] You can also use `nullify` on a newly born pointer to change its status from undefined to disassociated; this allows you to test the status with the `associated` inquiry function. [M&R, §6.5.4] (While many compilers will not produce a run-time error if you test an undefined pointer with `associated`, you can't rely on this *laissez-faire* in your programming.)

The initial status of an allocatable array is "not currently allocated." Its status changes to "allocated" when you give it storage with `allocate`, and back to "not currently allocated" when you use `deallocate`. [M&R, §6.5.1] You can test the status with the `allocated` inquiry function. Note that while you can also give a pointer fresh storage with `allocate`, you can't test this with `allocated` — only `associated` is allowed with pointers. Note also that nullifying an allocated pointer leaves its associated storage in limbo. You must instead `deallocate`, which gives the pointer a testable "disassociated" status.

While allocating an array that is already allocated gives an error, you are allowed to allocate a pointer that already has a target. This breaks the old association, and could leave the old target inaccessible if there is no other pointer associated with it. [M&R, §6.5.2] Deallocating an array or pointer that has not been allocated is always an error.

Allocated arrays that are local to a subprogram acquire the "undefined" status on exit from the subprogram unless they have the `SAVE` attribute. (Again, not all compilers enforce this, but be warned!) Such undefined arrays cannot be referenced in any way, so you should explicitly deallocate all allocated arrays that are not saved before returning from a subprogram. [M&R, §6.5.1] The same rule applies to arrays declared in modules that are currently accessed only by the subprogram. While you can reference undefined pointers (e.g., by first nullifying them), it is good programming practice to deallocate explicitly any allocated pointers declared locally before leaving a subprogram or module.

Now let's turn to using these features in programs. The simplest example is when we want to implement global storage of an array that needs to be accessed by two or more different routines, and we want the size of the array to be determined at run time. As mentioned earlier, we implement global storage with a `MODULE` rather than a `COMMON` block. (We ignore here the additional possibility of passing

global variables by having one routine CONTAINed within the other.) There are two good ways of handling the dynamical allocation in a MODULE. Method 1 uses an allocatable array:

```
MODULE a
REAL(SP), DIMENSION(:), ALLOCATABLE :: x
END MODULE a

SUBROUTINE b(y)
USE a
REAL(SP), DIMENSION(:) :: y
...
allocate(x(size(y)))
... [other routines using x called here] ...
END SUBROUTINE b
```

Here the global variable x gets assigned storage in subroutine b (in this case, the same as the length of y). The length of y is of course defined in the procedure that calls b. The array x is made available to any other subroutine called by b by including a USE a statement. The status of x can be checked with an allocated inquiry function on entry into either b or the other subroutine if necessary. As discussed above, you must be sure to deallocate x before returning from subroutine b. If you want x to retain its values between calls to b, you add the SAVE attribute to its declaration in a, and *don't* deallocate it on returning from b. (Alternatively, you could put a USE a in your main program, but we consider that bug-prone, since forgetting to do so can create all manner of difficult-to-diagnose havoc.) To avoid allocating x more than once, you test it on entry into b:

```
if (.not. allocated(x)) allocate(x(size(y)))
```

The second way to implement this type of global storage (Method 2) uses a pointer:

```
MODULE a
REAL(SP), DIMENSION(:), POINTER :: x
END MODULE a

SUBROUTINE b(y)
USE a
REAL(SP), DIMENSION(:) :: y
REAL(SP), DIMENSION(size(y)), TARGET :: xx
...
x=>xx
... [other routines using x called here] ...
END SUBROUTINE b
```

Here the *automatic array* xx gets its temporary storage automatically on entry into b, and automatically gets deallocated on exit from b. [M&R, §6.4] The global pointer x can access this storage in any routine with a USE a that is called by b. You can check that things are in order in such a called routine by testing x with associated. If you are going to use x for some other purpose as well, you should nullify it on leaving b so that it doesn't have undefined status. Note that this implementation does not allow values to be saved between calls: You can't SAVE automatic arrays — that's not what they're for. You would have to SAVE x in the module, and allocate it in the subroutine instead of pointing it to a suitable automatic array. But this is essentially Method 1 with the added complication of using a pointer, so Method 1 is simpler when you want to save values. When you don't

need to save values between calls, we lean towards Method 2 over Method 1 because we like the automatic allocation and deallocation, but either method works fine.

An example of Method 1 (allocatable array) is in routine `rkdumb` on page 1297. An example of Method 1 with `SAVE` is in routine `pwtset` on p. 1265. Method 2 (pointer) shows up in routines `newt` (p. 1196), `broydn` (p. 1199), and `fitexy` (p. 1286). A variation is shown in routines `linmin` (p. 1211) and `dlinmin` (p. 1212): When the array that needs to be shared is an argument of one of the routines, Method 2 is better.

An extension of these ideas occurs if we allocate some storage for an array initially, but then might need to increase the size of the array later without losing the already-stored values. The function `reallocate` in our utility module `nrutil` will handle this for you, but it expects a pointer argument as in Method 2. Since no automatic arrays are used, you are free to `SAVE` the pointer if necessary. Here is a simple example of how to use `reallocate` to create a workspace array that is local to a subroutine:

```
SUBROUTINE a
USE nrutil, ONLY : reallocate
REAL(SP), DIMENSION(:), POINTER, SAVE :: wksp
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
   init=.false.
   nullify(wksp)
   wksp=>reallocate(wksp,100)
end if
...
if (nterm > size(wksp)) wksp=>reallocate(wksp,2*size(wksp))
...
END SUBROUTINE a
```

Here the workspace is initially allocated a size of 100. If the number of elements used (`nterm`) ever exceeds the size of the workspace, the workspace is doubled. (In a realistic example, one would of course check that the doubled size is in fact big enough.) Fortran 90 experts can note that the `SAVE` on `init` is not strictly necessary: Any local variable that is initialized is automatically saved. [M&R, §7.5]

You can find similar examples of `reallocate` (with some further discussion) in `eulsum` (p. 1070), `hufenc` (p. 1348), and `arcode` (p. 1350). Examples of reallocate used with global variables in modules are in `odeint` (p. 1300) and `ran_state` (p. 1144).

Another situation where we have to use pointers and not allocatable arrays is when the storage is required for components of a derived type, which are not allowed to have the allocatable attribute. Examples are in `hufmak` (p. 1346) and `arcmak` (p. 1349).

Turning away from issues relating to global variables, we now consider several other important programming situations that are nicely handled with pointers. The first case is when we want a subroutine to return an array whose size is not known in advance. Since dummy arguments are not allocatable, we must use a pointer. Here is the basic construction:

```
SUBROUTINE a(x,nx)
REAL(SP), DIMENSION(:), POINTER :: x
INTEGER(I4B), INTENT(OUT) :: nx
LOGICAL(LGT), SAVE :: init=.true.
if (init) then
```

```
   init=.false.
   nullify(x)
else
   if (associated(x)) deallocate(x)
end if
...
nx=...
allocate(x(nx))
x(1:nx)=...
END SUBROUTINE a
```

Since the length of x can be found from `size(x)`, it is not absolutely necessary to pass `nx` as an argument. Note the use of the initial logic to avoid memory leaks. If a higher-level subroutine wants to recover the memory associated with `x` from the last call to `SUBROUTINE a`, it can do so by first deallocating it, and then nullifying the pointer. Examples of this structure are in `zbrak` (p. 1184), `period` (p. 1258), and `fasper` (p. 1259). A related situation is where we want a function to return an array whose size is not predetermined, such as in `voltra` on (p. 1326). The discussion of `voltra` also explains the potential pitfalls of functions returning pointers to dynamically allocated arrays.

A final useful pointer construction enables us to set up a data structure that is essentially an array of arrays, independently allocatable on each part. We are not allowed to declare an array of pointers in Fortran 90, but we can do this indirectly by defining a derived type that consists of a pointer to the appropriate kind of array. [M&R, §6.11] We can then define a variable that is an allocatable array of the new type. For example,

```
TYPE ptr_to_arr
   REAL(SP), DIMENSION(:), POINTER :: arr
END TYPE
TYPE(ptr_to_arr), DIMENSION(:), ALLOCATABLE :: x
...
allocate(x(n))
...
do i=1,n
   allocate(x(i)%arr(m))
end do
```

sets up a set `x` of `n` arrays of length `m`. See also the example in `mglin` (p. 1334).

There is a potential problem with dynamical memory allocation that we should mention. The Fortran 90 standard does not require that the compiler perform "garbage collection," that is, it is not required to recover deallocated memory into nice contiguous pieces for reuse. If you enter and exit a subroutine many times, and each time a large chunk of memory gets allocated and deallocated, you could run out of memory with a "dumb" compiler. You can often alleviate the problem by deallocating variables in the reverse order that you allocated them. This tends to keep a large contiguous piece of memory free at the top of the heap.

Scope, Visibility, and Data Hiding

An important principle of good programming practice is *modularization*, the idea that different parts of a program should be insulated from each other as much as possible. An important subcase of modularization is *data hiding*, the principle that actions carried out on variables in one part of the code should not be able to

affect the values of variables in other parts of the code. When it is necessary for one "island" of code to communicate with another, the communication should be through a well-defined interface that makes it obvious exactly what communication is taking place, and prevents any other interchange from occurring. Otherwise, different sections of code should not have access to variables that they don't need.

The concept of data hiding extends not only to variables, but also to the names of procedures that manipulate the variables: A program for screen graphics might give the user access to a routine for drawing a circle, but it might "hide" the names (and methods of operation) of the primitive routines used for calculating the coordinates of the points on the circumference. Besides producing code that is easier to understand and to modify, data hiding prevents unintended side effects from producing hard-to-find errors.

In Fortran, the principal language construction that effects data hiding is the use of subroutines. If all subprograms were restricted to have no more than ten executable statements per routine, and to communicate between routines only by an explicit list of arguments, the number of programming errors might be greatly reduced! Unfortunately few tasks can be easily coded in this style. For this and other reasons, we think that too much procedurization is a bad thing; one wants to find the *right* amount. Fortunately Fortran 90 provides several additional tools to help with data hiding.

Global variables and routine names are important, but potentially dangerous, things. In Fortran 90, global variables are typically encapsulated in modules. Access is granted only to routines with an appropriate `USE` statement, and can be restricted to specific identifiers by the `ONLY` option. [M&R, §7.10] In addition, variable and routine names within the module can be designated as `PUBLIC` or `PRIVATE` (see, e.g., `quad3d` on p. 1065). [M&R, §7.6]

The other way global variables get communicated is by having one routine `CONTAIN`ed within another. [M&R, §5.6] This usage is potentially lethal, however, because *all* the outer routine's variables are visible to the inner routine. You can try to control the problem somewhat by passing some variables back and forth as arguments of the inner routine, but that still doesn't prevent inadvertent side effects. (The most common, and most stupid, is inadvertent reuse of variables named `i` or `j` in the `CONTAIN`ed routine.) Also, a long list of arguments reduces the convenience of using an internal routine in the first place. We advise that internal subprograms be used with caution, and only to carry out simple tasks.

There are some good ways to use `CONTAINS`, however. Several of our recipes have the following structure: A principal routine is invoked with several arguments. It calls a subsidiary routine, which needs to know some of the principal routine's arguments, some global variables, and some values communicated directly as arguments to the subsidiary routine. In Fortran 77, we have usually coded this by passing the global variables in a `COMMON` block and all other variables as arguments to the subsidiary routine. If necessary, we copied the arguments of the primary routine before passing them to the subsidiary routine. In Fortran 90, there is a more elegant way of accomplishing this, as follows:

```
SUBROUTINE recipe(arg)
REAL(SP) :: arg
REAL(SP) :: global_var
call recipe_private
CONTAINS
```

```
SUBROUTINE recipe_private
...
call subsidiary(local_arg)
...
END SUBROUTINE recipe_private

SUBROUTINE subsidiary(local_arg)
...
END SUBROUTINE subsidiary
END SUBROUTINE recipe
```

Notice that the principal routine (`recipe`) has practically nothing in it — only declarations of variables intended to be visible to the subsidiary routine (`subsidiary`). All the real work of `recipe` is done in `recipe_private`. This latter routine has visibility on all of `recipe`'s variables, while any additional variables that `recipe_private` defines are *not* visible to `subsidiary` — which is the whole purpose of this way of organizing things. Obviously `arg` and `global_var` can be much more general data types than the example shown here, including function names. For examples of this construction, see `amoeba` (p. 1208), `amebsa` (p. 1222), `mrqmin` (p. 1292), and `medfit` (p. 1294).

Recursion

A subprogram is recursive if it calls itself. While forbidden in Fortran 77, recursion is allowed in Fortran 90. [M&R, §5.16–§5.17] You must supply the keyword `RECURSIVE` in front of the `FUNCTION` or `SUBROUTINE` keyword. In addition, if a `FUNCTION` calls itself directly, as opposed to calling another subprogram that in turn calls it, you must supply a variable to hold the result with the `RESULT` keyword. Typical syntax for this case is:

```
RECURSIVE FUNCTION f(x) RESULT(g)
REAL(SP) :: x,g
if ...
   g=...
else
   g=f(...)
end if
END FUNCTION f
```

When a function calls itself directly, as in this example, there always has to be a "base case" that does not call the function; otherwise the recursion never terminates. We have indicated this schematically with the `if...else...end if` structure.

On serial machines we tend to avoid recursive implementations because of the additional overhead they incur at execution time. Occasionally there are algorithms for which the recursion overhead is relatively small, and the recursive implementation is simpler than an iterative version. Examples in this book are `quad_3d` (p. 1065), `miser` (p. 1164), and `mglin` (p. 1334). Recursion is much more important when parallelization is the goal. We will encounter in Chapter 22 numerous examples of algorithms that can be parallelized with recursion.

SAVE Usage Style

A quirk of Fortran 90 is that any variable with initial values acquires the `SAVE` attribute automatically. [M&R, §7.5 and §7.9] As a help to understanding

an algorithm, we have elected to put an explicit SAVE on all variables that really do need to retain their values between calls to a routine. We do this even if it is redundant because the variables are initialized. Note that we generally prefer to assign initial values with initialization expressions rather than with DATA statements. We reserve DATA statements for cases where it is convenient to use the repeat count feature to set multiple occurrences of a value, or when binary, octal, or hexadecimal constants are used. [M&R, §2.6.1]

Named Control Structures

Fortran 90 allows control structures such as do loops and if blocks to be named. [M&R, §4.3–§4.5] Typical syntax is

```
name:do i=1,n
   ...
end do name
```

One use of naming control structures is to improve readability of the code, especially when there are many levels of nested loops and if blocks. A more important use is to allow exit and cycle statements, which normally refer to the innermost do loop in which they are contained, to transfer execution to the end of some outer loop. This is effected by adding the name of the outer loop to the statement: exit name or cycle name.

There is great potential for misuse with named control structures, since they share some features of the much-maligned goto. We recommend that you use them sparingly. For a good example of their use, contrast the Fortran 77 version of simplx with the Fortran 90 version on p. 1216.

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

21.6 And Coming Soon: Fortran 95

One of the more positive effects of Fortran 90's long gestation period has been the general recognition, both by the X3J3 committee and by the community at large, that Fortran needs to evolve over time. Indeed, as we write, the process of bringing forth a minor, but by no means insignificant, updating of Fortran 90 — named Fortran 95 — is well under way.

Fortran 95 will differ from Fortran 90 in about a dozen features, only a handful of which are of any importance to this book. Generally these are extensions that will make programming, especially parallel programming, easier. In this section we give a summary of the anticipated language changes. In §22.1 and §22.5 we will comment further on the implications of Fortran 95 to some parallel programming tasks; in §23.7 we comment on what differences Fortran 95 will make to our nrutil utility functions.

No programs in Chapters B1 through B20 of this book edition use any Fortran 95 extensions.

FORALL Statements and Blocks

Fortran 95 introduces a new `forall` control structure, somewhat akin to the `where` construct, but allowing for greater flexibility. It is something like a do-loop, but with the proviso that the indices looped over are allowed to be done in any order (ideally, in parallel). The `forall` construction comes in both single-statement and block variants. Instead of using the do-loop's comma-separated triplets of lower-value, upper-value, and increment, it borrows its syntax from the colon-separated form of array sections. Some examples will give you the idea.

Here is a simple example that could alternatively be done with Fortran 90's array sections and `transpose` intrinsic:

```
forall (i=1:20, j=1:10:2) x(i,j)=y(j,i)
```

The block form allows more than one executable statement:

```
forall (i=1:20, j=1:10:2)
   x(i,j)=y(j,i)
   z(i,j)=y(i,j)**2
end forall
```

Here is an example that cannot be done with Fortran 90 array sections:

```
forall (i=1:20, j=1:20) a(i,j)=3*i+j**2
```

`forall` statements can also take optional masks that restrict their action to a subset of the loop index combinations:

```
forall (i=1:100, j=1:100, (i>=j .and. x(i,j)/=0.0) ) x(i,j)=1.0/x(i,j)
```

`forall` constructions can be nested, or nested inside `where` blocks, or have `where` constructions inside them. An additional new feature in Fortran 95 is that `where` blocks can themselves be nested.

PURE Procedures

Because the inside iteration of a `forall` block can be done in any order, or in parallel, there is a logical difficulty in allowing functions or subroutines inside such blocks: If the function or subroutine has *side effects* (that is, if it changes any data elsewhere in the machine, or in its own saved variables) then the result of a `forall` calculation could depend on the order in which the iterations happen to be done. This can't be tolerated, of course; hence a new `PURE` attribute for subprograms.

While the exact stipulations are somewhat technical, the basic idea is that if you declare a function or subroutine as `PURE`, with a syntax like,

```
PURE FUNCTION myfunc(x,y,z)
```

or

```
PURE SUBROUTINE mysub(x,y,z)
```

then you are guaranteeing to the compiler (and it will enforce) that the only values changed by `mysub` or `myfunc` are returned function values, subroutine arguments with the `INTENT(OUT)` attribute, and automatic (scratch) variables within the procedure.

You can then use your pure procedures within `forall` constructions. Pure functions are also allowed in some specification statements.

ELEMENTAL Procedures

Fortran 95 removes Fortran 90's nagging restriction that only intrinsic functions are elemental. The way this works is that you write a pure procedure that operates on scalar values, but include the attribute `ELEMENTAL` (which automatically implies `PURE`). Then, as long as the function has an explicit interface in the referencing program, you can call it with any shape of argument, and it will act elementally. Here's an example:

```
ELEMENTAL FUNCTION myfunc(x,y,z)
REAL :: x,y,z,myfunc
...
myfunc = ...
END
```

In a program with an explicit interface for `myfunc` you could now have

```
REAL, DIMENSION(10,20) :: x,y,z,w
...
w=myfunc(x,y,z)
```

Pointer and Allocatable Improvements

Fortran 95, unlike Fortran 90, requires that any allocatable variables (except those with `SAVE` attributes) that are allocated within a subprogram be automatically deallocated by the compiler when the subprogram is exited. This will remove Fortran 90's "undefined allocation status" bugaboo.

Fortran 95 also provides a method for pointer variables to be born with disassociated association status, instead of the default (and often inconvenient) "undefined" status. The syntax is to add an initializing `=> NULL()` to the declaration, as:

```
REAL, DIMENSION(:,:), POINTER :: mypoint => NULL()
```

This does not, however, eliminate the possibility of undefined association status, because you have to remember to use the null initializer if want your pointer to be disassociated.

Some Other Fortran 95 Features

In Fortran 95, `maxloc` and `minloc` have the additional optional argument `DIM`, which causes them to act on all one-dimensional sections that span through the named dimension. This provides a means for getting the locations of the values returned by the corresponding functions `maxval` and `minval` in the case that their `DIM` argument is present.

The `sign` intrinsic can now distinguish a negative from a positive real zero value: `sign(2.0,-0.0)` is -2.0.

There is a new intrinsic subroutine `cpu_time(time)` that returns as a real value `time` a process's elapsed CPU time.

There are some minor changes in the namelist facility, in defining minimum field widths for the I, B, O, Z, and F edit descriptors, and in resolving minor conflicts with some other standards.

Chapter 22. Introduction to Parallel Programming

22.0 Why Think Parallel?

In recent years we Numerical Recipes authors have increasingly become convinced that a certain revolution, cryptically denoted by the words "parallel programming," is about to burst forth from its gestation and adolescence in the community of supercomputer users, and become the mainstream methodology for all computing.

Let's review the past: Take a screwdriver and open up the computer (workstation or PC) that sits on your desk. (Don't blame us if this voids your warranty; and be sure to unplug it first!) Count the integrated circuits — just the bigger ones, with more than a million gates (transistors). As we write, in 1995, even lowly memory chips have one or four million gates, and this number will increase rapidly in coming years. You'll probably count at least dozens, and often hundreds, of such chips in your computer.

Next ask, how many of these chips are CPUs? That is, how many implement von Neumann processors capable of executing arbitrary, stored program code? For most computers, in 1995, the answer is: about one. A significant number of computers do have secondary processors that offload input-output and/or video functions. So, two or three is often a more accurate answer, but only one is usually under the user's direct control.

Why do our desktop computers have dozens or hundreds of memory chips, but most often only one (user-accessible) CPU? Do CPU chips intrinsically cost more to manufacture? No. Are CPU chips more expensive than memory chips? Yes, primarily because fixed development and design costs must be distributed over a smaller number of units sold. We have been in a kind of economic equilibrium: CPU's are relatively expensive because there is only one per computer; and there is only one per computer, because they are relatively expensive.

Stabilizing this equilibrium has been the fact that there has been no standard, or widely taught, methodology for parallel programming. Except for the special case of scientific computing on supercomputers (where large problems often have a regular or geometric character), it is not too much of an exaggeration to say that nobody *really knows how* to program multiprocessor machines. Symmetric multiprocessor

operating systems, for example, have been very slow in developing; and efficient, parallel methodologies for query-serving on large databases are even now a subject of continuing research.

However, things are now changing. We consider it an easy prognostication that, by the first years of the new century, the typical desktop computer will have 4 to 8 user-accessible CPUs; ten years after that, the typical number will be between 16 and 512. It is not coincidence that these numbers are characteristic of supercomputers (including some quite different architectures) in 1995. The rough rule of ten years' lag from supercomputer to desktop has held firm for quite some time now.

Scientists and engineers have the advantage that techniques for parallel computation in their disciplines *have* already been developed. With multiprocessor workstations right around the corner, we think that now is the right time for scientists and engineers who use computers to start *thinking parallel*. We don't mean that you should put an axe through the screen of your fast serial (single-CPU) workstation. We do mean, however, that you should start programming somewhat differently on that workstation, indeed, start thinking a bit differently about the way that you approach numerical problems in general.

In this volume of *Numerical Recipes in Fortran*, our pedagogical goal is to show you that there are conceptual and practical benefits in parallel thinking, even if you are using a serial machine today. These benefits include conciseness and clarity of code, reusability of code in wider contexts, and (not insignificantly) increased portability of code to today's parallel supercomputers. Of course, on parallel machines, either supercomputers today or desktop machines tomorrow, the benefits of thinking parallel are much more tangible: They translate into significant improvements in efficiency and computational capability.

Thinking Parallel with Fortran 90

Until very recently, a strong inhibition to thinking parallel was the lack of any standard, architecture-independent, computer language in which to think. That has changed with the finalization of the Fortran 90 language standard, and with the availability of good, optimizing Fortran 90 compilers on a variety of platforms.

There is a significant body of opinion (with which we, however, disagree) that there is no such thing as architecture-independent parallel programming. Proponents of this view, who are generally committed wizards at programming on one or another particular architecture, point to the fact that algorithms that are optimized to one architecture can run hundreds of times more slowly on other architectures. And, they are correct!

Our opposing point of view is one of pragmatism. We think that it is not hard to learn, in a general way, what kinds of architectures are in general use, and what kinds of parallel constructions work well (or poorly) on each kind. With this knowledge (much of which we hope to develop in this book) the user can, we think, write good, general-purpose parallel code that works on a variety of architectures — including, importantly, on purely serial machines. Equally important, the user will be aware of when certain parts of a code can be significantly improved on some, but not other, architectures.

Fortran 90 is a good test-bench for this point of view. It is not the perfect language for parallel programming. But it is *a* language, and it is the only

cross-platform *standard* language now available. The committee that developed the language between 1978 and 1991 (known technically as X3J3) had strong representation from both a traditional "vectorization" viewpoint (e.g., from the Cray XMP and YMP series of computers), and also from the "data parallel" or "SIMD" viewpoints of parallel machines like the CM-2 and CM-5 from Thinking Machines, Inc. Language compromises were made, and a few (in our view) almost essential features were left out (see §22.5). But, by and large, the necessary tools are there: If you learn to think parallel in Fortran 90, you will easily be able to transfer the skill to future parallel standards, whether they are Fortran-based, C-based, or other.

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

22.1 Fortran 90 Data Parallelism: Arrays and Intrinsics

The underlying model for parallel computation in Fortran 90 is *data parallelism*, implemented by the use of arrays of data, and by the provision of operations and intrinsic functions that act on those arrays in parallel, in a manner optimized by the compiler for each particular hardware architecture. We will not try to draw a fine definitional distinction between "data parallelism" and so-called SIMD (single instruction multiple data) programming. For our purposes the two terms mean about the same thing: The programmer writes a single operation, "+" say, and the compiler causes it to be carried out on multiple pieces of data in as parallel a manner as the underlying hardware allows.

Any kind of parallel computing that is not SIMD is generally called MIMD (multiple instruction multiple data). A parallel programming language with MIMD features might allow, for example, several different subroutines — acting on different parts of the data — to be called into execution simultaneously. Fortran 90 has few, if any, MIMD constructions. A Fortran 90 compiler might, on some machines, execute MIMD code in implementing some Fortran 90 intrinsic functions (`pack` or `unpack`, e.g.), but this will be hidden from the Fortran 90 user. Some extensions of Fortran 90, like HPF, do implement MIMD features explicitly; but we will not consider these in this book. Fortran 95's `forall` and `PURE` extensions (see §21.6) will allow some significantly greater access to MIMD features (see §22.5).

Array Parallel Operations

We have already met the most basic, and most important, parallel facility of Fortran 90, namely, the ability to use whole arrays in expressions and assignments, with the indicated operations being effected in parallel across the array. Suppose, for example, we have the two-dimensional matrices `a`, `b`, and `c`,

```
REAL, DIMENSION(30,30) :: a,b,c
```

Then, instead of the serial construction,

```
do j=1,30
   do k=1,30
      c(j,k)=a(j,k)+b(j,k)
   end do
end do
```

which is of course perfectly valid Fortran 90 code, we can simply write

```
c=a+b
```

The compiler deduces from the declaration statement that `a`, `b`, and `c` are matrices, and what their bounding dimensions are.

Let us dwell for a moment on the conceptual differences between the serial code and parallel code for the above matrix addition. Although one is perhaps used to seeing the nested do-loops as simply an idiom for "do-the-enclosed-on-all-components," it in fact, according to the rules of Fortran, specifies a very particular time-ordering for the desired operations. The matrix elements are added by rows, in order (`j=1,30`), and within each row, by columns, in order (`k=1,30`).

In fact, the serial code above *overspecifies* the desired task, since it is guaranteed by the laws of mathematics that the order in which the element operations are done is of no possible relevance. Over the 50 year lifetime of serial von Neuman computers, we programmers have been brainwashed to break up all problems into single executable streams *in the time dimension only*. Indeed, the major design problem for supercomputer compilers for the last 20 years has been to *undo* such serial constructions and recover the underlying "parallel thoughts," for execution in vector or parallel processors. Now, rather than taking this expensive detour into and out of serial-land, we are asked simply to say what we mean in the first place, `c=a+b`.

The essence of parallel programming is *not* to force "into the time dimension" (i.e., to serialize) operations that naturally extend across a span of data, that is, "in the space dimension." If it were not for 50-year-old collective habits, and the languages designed to support them, parallel programming would probably strike us as more natural than its serial counterpart.

Broadcasts and Dimensional Expansion: SSP vs. MMP

We have previously mentioned the Fortran 90 rule that a scalar variable is conformable with any shape array. Thus, we can implement a calculation such as

$$y_i = x_i + s, \qquad i = 1, \ldots, n \tag{22.1.1}$$

with code like

```
y=x+s
```

where we of course assume previous declarations like

```
REAL(SP) :: s
REAL(SP), DIMENSION(n) :: x,y
```

with `n` a compile-time constant or dummy argument. (Hereafter, we will omit the declarations in examples that are this simple.)

This seemingly simple construction actually hides an important underlying parallel capability, namely, that of *broadcast*. The sums in `y=x+s` are done in parallel

on different CPUs, each CPU accessing different components of `x` and `y`. Yet, they all must access the same scalar value `s`. If the hardware has local memory for each CPU, the value of `s` must be replicated and transferred to each CPU's local memory. On the other hand, if the hardware implements a single, global memory space, it is vital to do something that mitigates the traffic jam potentially caused by all the CPUs trying to access the same memory location at the same time. (We will use the term "broadcast" to refer equally to both cases.) Although hidden from the user, Fortran 90's ability to do broadcasts is an essential feature of it as a parallel language.

Broadcasts can be more complicated than the above simple example. Consider, for example, the calculation

$$w_i = \sum_{j=1}^{n} |x_i + x_j|, \qquad i = 1, \ldots, n \tag{22.1.2}$$

Here, we are doing n^2 operations: For each of n values of i there is a sum over n values of j.

Serial code for this calculation might be

```
do i=1,n
   w(i)=0.
   do j=1,n
      w(i)=w(i)+abs(x(i)+x(j))
   end do
end do
```

The obvious immediate parallelization in Fortran 90 uses the `sum` intrinsic function to eliminate the inner do-loop. This would be a suitable amount of parallelization for a small-scale parallel machine, with a few processors:

```
do i=1,n
   w(i)=sum(abs(x(i)+x))
end do
```

Notice that the conformability rule implies that a new value of `x(i)`, a scalar, is being broadcast to all the processors involved in the `abs` and `sum`, with each iteration of the loop over `i`.

What about the outer do-loop? Do we need, or want, to eliminate it, too? That depends on the architecture of your computer, and on the tradeoff between time and memory in your problem (a common feature of all computing, no less so parallel computing). Here is an implementation that is free of all do-loops, in principle capable of being executed in a small number (independent of n) of parallel operations:

```
REAL(SP), DIMENSION(n,n) :: a
...
a = spread(x,dim=2,ncopies=n)+spread(x,dim=1,ncopies=n)
w = sum(abs(a),dim=1)
```

This is an example of what we call *dimensional expansion*, as implemented by the `spread` intrinsic. Although the above may strike you initially as quite a cryptic construction, it is easy to learn to read it. In the first assignment line, a matrix is constructed with all possible values of `x(i)+x(j)`. In the second assignment line, this matrix is collapsed back to a vector by applying the sum operation to the absolute value of its elements, across one of its dimensions.

More explicitly, the first line creates a matrix `a` by adding two matrices each constructed via `spread`. In `spread`, the `dim` argument specifies which argument is *duplicated*, so that the first term *varies* across its first (row) dimension, and vice versa for the second term:

$$a_{ij} = x_i + x_j$$
$$= \begin{pmatrix} x_1 & x_1 & x_1 & \dots \\ x_2 & x_2 & x_2 & \dots \\ x_3 & x_3 & x_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} + \begin{pmatrix} x_1 & x_2 & x_3 & \dots \\ x_1 & x_2 & x_3 & \dots \\ x_1 & x_2 & x_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{22.1.3}$$

Since equation (22.1.2) above is symmetric in i and j, it doesn't really matter what value of `dim` we put in the `sum` construction, but the value `dim=1` corresponds to summing across the rows, that is, down each column of equation (22.1.3).

Be sure that you understand that the `spread` construction changed an $O(n)$ memory requirement into an $O(n^2)$ one! If your values of n are large, this is an impossible burden, and the previous implementation with a single do-loop remains the only practical one. On the other hand, if you are working on a massively parallel machine, whose number of processors is comparable to n^2 (or at least much larger than n), then the `spread` construction, and the underlying broadcast capability that it invokes, leads to a big win: All n^2 operations can be done in parallel. This distinction between small-scale parallel machines — which we will hereafter refer to as *SSP machines* — and massively multiprocessor machines — which we will refer to as *MMP machines* — is an important one. A main goal of parallelism is to saturate the available number of processors, and algorithms for doing so are often different in the SSP and MMP opposite limits. Dimensional expansion is one method for saturating processors in the MMP case.

Masks and "Index Loss"

An instructive extension of the above example is the following case of a product that omits one term (the diagonal one):

$$w_i = \prod_{\substack{j=1 \\ j \neq i}}^{n} (x_j - x_i), \qquad i = 1, \dots, n \tag{22.1.4}$$

Formulas like equation (22.1.4) frequently occur in the context of interpolation, where all the x_i's are known to be distinct, so let us for the moment assume that this is the case.

Serial code for equation (22.1.4) could be

```
do i=1,n
   w(i)=1.0_sp
   do j=1,n
      if (j /= i) w(i)=w(i)*(x(j)-x(i))
   end do
end do
```

Parallel code for SSP machines, or for large enough n on MMP machines, could be

```
do i=1,n
   w(i)=product( x-x(i), mask=(x/=x(i)) )
end do
```

Here, the `mask` argument in the `product` intrinsic function causes the diagonal term to be omitted from the product, as we desire. There are some features of this code, however, that bear commenting on.

First, notice that, according to the rules of conformability, the expression `x/=x(i)` broadcasts the scalar `x(i)` and generates a logical array of length n, suitable for use as a `mask` in the `product` intrinsic. It is quite common in Fortran 90 to generate masks "on the fly" in this way, particularly if the mask is to be used only once.

Second, notice that the `j` index has disappeared completely. It is now implicit in the two occurrences of `x` (equivalent to `x(1:n)`) on the right-hand side. With the disappearance of the `j` index, we also lose the ability to do the test on `i` and `j`, but must use, in essence, `x(i)` and `x(j)` instead! That is a very general feature in Fortran 90: when an operation is done in parallel across an array, there is *no associated index* available within the operation. This "index loss," as we will see in later discussion, can sometimes be quite an annoyance.

A language construction present in CM [Connection Machine] Fortran, the so-called `forall`, which would have allowed access to an associated index in many cases, was eliminated from Fortran 90 by the X3J3 committee, in a controversial decision. Such a construction will come into the language in Fortran 95.

What about code for an MMP machine, where we are willing to use dimensional expansion to achieve greater parallelism? Here, we can write,

```
a = spread(x,dim=2,ncopies=n)-spread(x,dim=1,ncopies=n)
w = product(a,dim=1,mask=(a/=0.))
```

This time it does matter that the value of `dim` in the `product` intrinsic is 1 rather than 2. If you write out the analog of equation (22.1.3) for the present example, you'll see that the above fragment is the right way around. The problem of index loss is still with us: we have to construct a mask from the array `a`, not from its indices, *both* of which are now lost to us!

In most cases, there are workarounds (more, or less, awkward as they may be) for the problem of index loss. In the worst cases, which are quite rare, you have to create objects to hold, and thus bring back into play, the lost indices. For example,

```
INTEGER(I4B), DIMENSION(n) :: jj
...
jj = (/ (i,i=1,n) /)
do i=1,n
   w(i)=product( x-x(i), mask=(jj/=i) )
end do
```

Now the array `jj` is filled with the "lost" j index, so that it is available for use in the mask. A similar technique, involving spreads of `jj`, can be used in the above MMP code fragment, which used dimensional expansion. (Fortran 95's `forall` construction will make index loss much less of a problem. See §21.6.)

Incidentally, the above Fortran 90 construction, `(/ (i,i=1,n) /)`, is called an *array constructor with implied do list*. For reasons to be explained in §22.2, we almost never use this construction, in most cases substituting a Numerical Recipes utility function for generating arithmetical progressions, which we call `arth`.

Interprocessor Communication Costs

It is both a blessing and a curse that Fortran 90 completely hides from the user the underlying machinery of interprocessor communication, that is, the way that data values computed by (or stored locally near) one CPU make their way to a different CPU that might need them next. The blessing is that, by and large, the Fortran 90 programmer need not be concerned with how this machinery works. If you write

```
a(1:10,1:10) = b(1:10,1:10) + c(10:1:-1,10:1:-1)
```

the required upside-down-and-backwards values of the array `c` are just *there*, no matter that a great deal of routing and switching may have taken place. An ancillary blessing is that this book, unlike so many other (more highly technical) books on parallel programming (see references below) need not be filled with complex and subtle discussions of CPU connectivity, topology, routing algorithms, and so on.

The curse is, just as you might expect, that the Fortran 90 programmer can't control the interprocessor communication, even when it is desirable to do so. A few regular communication patterns are "known" to the compiler through Fortran 90 intrinsic functions, for example `b=transpose(a)`. These, presumably, are done in an optimal way. However, many other regular patterns of communication, which might also allow highly optimized implementations, don't have corresponding intrinsic functions. (An obvious example is the "butterfly" pattern of communication that occurs in fast Fourier transforms.) These, if coded in Fortran 90 by using general vector subscripts (e.g., `barr=arr(iarr)` or `barr(jarr)=arr`, where `iarr` and `jarr` are integer arrays), lose all possibility of being optimized. The compiler can't distinguish a communication step with regular structure from one with general structure, so it must assume the worst case, potentially resulting in very slow execution.

About the only thing a Fortran 90 programmer can do is to start with a general awareness of the kind of apparently parallel constructions that *might* be quite slow on his/her parallel machine, and then to refine that awareness by actual experience and experiment. Here is our list of constructions most likely to cause interprocessor communication bottlenecks:

- vector subscripts, like `barr=arr(iarr)` or `barr(jarr)=arr` (that is, general gather/scatter operations)
- the `pack` and `unpack` intrinsic functions
- mixing positive strides and negative strides in a single expression (as in the above `b(1:10,1:10)+c(10:1:-1,10:1:-1)`)
- the `reshape` intrinsic when used with the `order` argument
- possibly, the `cshift` and `eoshift` extrinsics, especially for nonsmall values of the shift.

On the other hand, the fact is that these constructions *are* parallel, and *are* there for you to use. If the alternative to using them is strictly serial code, you should almost always give them a try.

Linear Algebra

You should be alert for opportunities to use combinations of the `matmul`, `spread`, and `dot_product` intrinsics to perform complicated linear algebra calculations. One useful intrinsic that is not provided in Fortran 90 is the *outer product*

of two vectors,

$$c_{ij} = a_i b_j \tag{22.1.5}$$

We already know how to implement this (cf. equation 22.1.3):

```
c = spread(a,dim=2,ncopies=size(b))*spread(b,dim=1,ncopies=size(a))
```

In fact, this operation occurs frequently enough to justify making it a utility function, `outerprod`, which we will do in Chapter 23. There we also define other "outer" operations between vectors, where the multiplication in the outer product is replaced by another binary operation, such as addition or division.

Here is an example of using these various functions: Many linear algebra routines require that a submatrix be updated according to a formula like

$$a_{jk} = a_{jk} + b_i a_{ji} \sum_{p=i}^{m} a_{pi} a_{pk}, \qquad j = i, \ldots, m, \quad k = l, \ldots, n \tag{22.1.6}$$

where i, m, l, and n are fixed values. Using an array slice like `a(:,i)` to turn a_{pi} into a vector indexed by p, we can code the sum with a `matmul`, yielding a vector indexed by k:

```
temp(l:n)=b(i)*matmul(a(i:m,i),a(i:m,l:n))
```

Here we have also included the multiplication by b_i, a scalar for fixed i. The vector `temp`, along with the vector a_{ji} = `a(:,i)`, is then turned into a matrix by the `outerprod` utility and used to increment a_{jk}:

```
a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),temp(l:n))
```

Sometimes the update formula is similar to (22.1.6), but with a slight permutation of the indices. Such cases can be coded as above if you are careful about the order of the quantities in the `matmul` and the `outerprod`.

CITED REFERENCES AND FURTHER READING:

Akl, S.G. 1989, *The Design and Analysis of Parallel Algorithms* (Englewood Cliffs, NJ: Prentice Hall).

Bertsekas, D.P., and Tsitsiklis, J.N. 1989, *Parallel and Distributed Computation: Numerical Methods* (Englewood Cliffs, NJ: Prentice Hall).

Carey, G.F. 1989, *Parallel Supercomputing: Methods, Algorithms, and Applications* (New York: Wiley).

Fountain, T.J. 1994, *Parallel Computing: Principles and Practice* (New York: Cambridge University Press).

Golub, G., and Ortega, J.M. 1993, *Scientific Computing: An Introduction with Parallel Computing* (San Diego, CA: Academic Press).

Fox, G.C., et al. 1988, *Solving Problems on Concurrent Processors*, Volume I (Englewood Cliffs, NJ: Prentice Hall).

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2* (Bristol and Philadelphia: Adam Hilger).

Kumar, V., et al. 1994, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms* (Redwood City, CA: Benjamin/Cummings).

Lewis, T.G., and El-Rewini, H. 1992, *Introduction to Parallel Computing* (Englewood Cliffs, NJ: Prentice Hall).

Modi, J.J. 1988, *Parallel Algorithms and Matrix Computation* (New York: Oxford University Press).

Smith, J.R. 1993, *The Design and Analysis of Parallel Algorithms* (New York: Oxford University Press).

Van de Velde, E. 1994, *Concurrent Scientific Computing* (New York: Springer-Verlag).

22.2 Linear Recurrence and Related Calculations

We have already seen that Fortran 90's *array constructor with implied do list* can be used to generate simple series of integers, like `(/ (i,i=1,n) /)`. Slightly more generally, one might want to generate an arithmetic progression, by the formula

$$v_j = b + (j-1)a, \qquad j = 1, \ldots, n \tag{22.2.1}$$

This is readily coded as

```
v(1:n) = (/ (b+(j-1)*a, j=1,n) /)
```

Although it is concise, and valid, *we don't like this coding.* The reason is that it violates the fundamental rule of "thinking parallel": it turns a parallel operation across a data vector into a serial do-loop over the components of that vector. Yes, we know that the compiler might be smart enough to generate parallel code for implied do lists; but it also might *not* be smart enough, here or in more complicated examples.

Equation (22.2.1) is also the simplest example of a *linear recurrence relation*. It can be rewritten as

$$v_1 = b, \qquad v_j = v_{j-1} + a, \qquad j = 2, \ldots, n \tag{22.2.2}$$

In this form (assuming that, in more complicated cases, one doesn't know an explicit solution like equation 22.2.1) one can't write an explicit array constructor. Code like

```
v(1) = b
v(2:n) = (/ (v(j-1)+a,j=2,n) /)      ! wrong
```

is legal Fortran 90 syntax, but illegal semantics; it does *not* do the desired recurrence! (The rules of Fortran 90 require that all the components of `v` on the right-hand side be evaluated before any of the components on the left-hand side are set.) Yet, as we shall see, techniques for accomplishing the evaluation in parallel are available.

With this as our starting point, we now survey some particular tricks of the (parallel) trade.

Subvector Scaling: Arithmetic and Geometric Progressions

For explicit arithmetic progressions like equation (22.2.1), the simplest parallel technique is *subvector scaling* [1]. The idea is to work your way through the desired vector in larger and larger parallel chunks:

$$\begin{aligned} v_1 &= b \\ v_2 &= b + a \\ v_{3\ldots4} &= v_{1\ldots2} + 2a \\ v_{5\ldots8} &= v_{1\ldots4} + 4a \\ v_{9\ldots16} &= v_{1\ldots8} + 8a \end{aligned} \tag{22.2.3}$$

And so on, until you reach the length of your vector. (The last step will not necessarily go all the way to the next power of 2, therefore.) The powers of 2, times a, can of course be obtained by successive doublings, rather than the explicit multiplications shown above.

You can see that subvector scaling requires about $\log_2 n$ parallel steps to process a vector of length n. Equally important for serial machines, or SSP machines, the scalar operation count for subvector scaling is no worse than entirely serial code: each new component v_i is produced by a single addition.

If addition is replaced by multiplication, the identical algorithm will produce geometric progressions, instead of arithmetic progressions. In Chapter 23, we will use subvector scaling to implement our utility functions `arth` and `geop` for these two progressions. (You can then call one of these functions instead of recoding equation 22.2.3 every time you need it.)

Vector Reduction: Evaluation of Polynomials

Logically related to subvector scaling is the case where a calculation can be parallelized across a vector that *shrinks* by a factor of 2 in each iteration, until a desired *scalar* result is reached. A good example of this is the parallel evaluation of a polynomial [2]

$$P(x) = \sum_{j=0}^{N} c_j x^j \tag{22.2.4}$$

For clarity we take the special case of $N = 5$. Start with the vector of coefficients (imagining appended zeros, as shown):

$$c_0, \quad c_1, \quad c_2, \quad c_3, \quad c_4, \quad c_5, \quad 0, \quad \ldots$$

Now, add the elements by pairs, multiplying the second of each pair by x:

$$c_0 + c_1 x, \quad c_2 + c_3 x, \quad c_4 + c_5 x, \quad 0, \quad \ldots$$

Now, the same operation, but with the multiplier x^2:

$$(c_0 + c_1 x) + (c_2 + c_3 x)x^2, \quad (c_4 + c_5 x) + (0)x^2, \quad 0, \quad \ldots$$

And a final time, with multiplier x^4:

$$[(c_0 + c_1x) + (c_2 + c_3x)x^2] + [(c_4 + c_5x) + (0)x^2]x^4, \quad 0, \quad \ldots$$

We are left with a vector of (active) length 1, whose value is the desired polynomial evaluation. (You can see that the zeros are just a bookkeeping device for taking account of the case where the active subvector has odd length.) The key point is that the combining by pairs is a parallel operation at each stage.

As in subvector scaling, there are about $\log_2 n$ parallel stages. Also as in subvector scaling, our total operations count is only negligibly different from purely scalar code: We do one add and one multiply for each original coefficient c_j. The only extra operations are $\log_2 n$ successive squarings of x; but this comes with the extra benefit of better roundoff properties than the standard scalar coding. In Chapter 23 we use vector reduction to implement our utility function `poly` for polynomial evaluation.

Recursive Doubling: Linear Recurrence Relations

Please don't confuse our use of the word "recurrence" (as in "recurrence relation," "linear recurrence," or equation 22.2.2) with the words "recursion" and "recursive," which both refer to the idea of a subroutine calling itself to obtain an efficient or concise algorithm. There are ample grounds for confusion, because recursive algorithms are in fact a good way of obtaining parallel solutions to linear recurrence relations, as we shall now see!

Consider the general first order linear recurrence relation

$$u_j = a_j + b_{j-1}u_{j-1}, \qquad j = 2, 3, \ldots, n \tag{22.2.5}$$

with initial value $u_1 = a_1$. On a serial machine, we evaluate such a recurrence with a simple do-loop. To parallelize the recurrence, we can employ the powerful general strategy of *recursive doubling*. Write down equation (22.2.5) for $2j$ and for $2j - 1$:

$$u_{2j} = a_{2j} + b_{2j-1}u_{2j-1} \tag{22.2.6}$$

$$u_{2j-1} = a_{2j-1} + b_{2j-2}u_{2j-2} \tag{22.2.7}$$

Substitute equation (22.2.7) in equation (22.2.6) to eliminate u_{2j-1} and get

$$u_{2j} = (a_{2j} + a_{2j-1}b_{2j-1}) + (b_{2j-2}b_{2j-1})u_{2j-2} \tag{22.2.8}$$

This is a new recurrence of the same form as (22.2.5) but over only the even u_j, and hence involving only $n/2$ terms. Clearly we can continue this process recursively, halving the number of terms in the recurrence at each stage, until we are left with a recurrence of length 1 or 2 that we can do explicitly. Each time we finish a subpart of the recursion, we fill in the odd terms in the recurrence, using equation (22.2.7). In practice, it's even easier than it sounds. Turn to Chapter B5 to see a straightforward implementation of this algorithm as the recipe `recur1`.

On a machine with more processors than n, all the arithmetic at each stage of the recursion can be done simultaneously. Since there are of order $\log n$ stages in the

recursion, the execution time is $O(\log n)$. The total number of operations carried out is of order $n + n/2 + n/4 + \cdots = O(n)$, the same as for the obvious serial do-loop.

In the utility routines of Chapter 23, we will use recursive doubling to implement the routines `poly_term`, `cumsum`, and `cumprod`. We *could* use recursive doubling to implement parallel versions of `arth` and `geop` (arithmetic and geometric progressions), and `zroots_unity` (complex nth roots of unity), but these can be done slightly more efficiently by subvector scaling, as discussed above.

Cyclic Reduction: Linear Recurrence Relations

There is a variant of recursive doubling, called *cyclic reduction*, that can be implemented with a straightforward iteration loop, instead of a recursive procedure call. [3] Here we start by writing down the recurrence (22.2.5) for *all* adjacent terms u_j and u_{j-1} (not just the even ones, as before). Eliminating u_{j-1}, just as in equation (22.2.8), gives

$$u_j = (a_j + a_{j-1}b_{j-1}) + (b_{j-2}b_{j-1})u_{j-2} \tag{22.2.9}$$

which is a first order recurrence with new coefficients a'_j and b'_j. Repeating this process gives successive formulas for u_j in terms of u_{j-2}, u_{j-4}, $u_{j-8}\ldots$. The procedure terminates when we reach u_{j-n} (for n a power of 2), which is zero for all j. Thus the last step gives u_j equal to the last set of a'_j's.

Here is a code fragment that implements cyclic reduction by direct iteration. The quantities a'_j are stored in the variable `recur1`.

```
recur1=a
bb=b
j=1
do
   if (j >= n) exit
   recur1(j+1:n)=recur1(j+1:n)+bb(j:n-1)*recur1(1:n-j)
   bb(2*j:n-1)=bb(2*j:n-1)*bb(j:n-j-1)
   j=2*j
enddo
```

In cyclic reduction the length of the vector u_j that is updated at each stage does *not* decrease by a factor of 2 at each stage, but rather only decreases from $\sim n$ to $\sim n/2$ during all $\log_2 n$ stages. Thus the total number of operations carried out is $O(n \log n)$, as opposed to $O(n)$ for recursive doubling. For a serial machine or SSP machine, therefore, cyclic reduction is rarely superior to recursive doubling when the latter can be used. For an MMP machine, however, the issue is less clear cut, because the pattern of communication in cyclic reduction is quite different (and, for some parallel architectures, possibly more favorable) than that of recursive doubling.

Second Order Recurrence Relations

Consider the second order recurrence relation

$$y_j = a_j + b_{j-2}y_{j-1} + c_{j-2}y_{j-2}, \qquad j = 3, 4, \ldots, n \tag{22.2.10}$$

with initial values

$$y_1 = a_1, \qquad y_2 = a_2 \tag{22.2.11}$$

Our labeling of subscripts is designed to make it easy to enter the coefficients in a computer program: You need to supply $a_1, \ldots, a_n$, $b_1, \ldots, b_{n-2}$, and $c_1, \ldots, c_{n-2}$. Rewrite the recurrence relation in the form ([3])

$$\begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix} = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix} \begin{pmatrix} y_{j-1} \\ y_j \end{pmatrix}, \qquad j = 2, \ldots, n-1 \tag{22.2.12}$$

that is,

$$\mathbf{u}_j = \mathbf{a}_j + \mathbf{b}_{j-1} \cdot \mathbf{u}_{j-1}, \qquad j = 2, \ldots, n-1 \tag{22.2.13}$$

where

$$\mathbf{u}_j = \begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix}, \quad \mathbf{a}_j = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix}, \quad \mathbf{b}_{j-1} = \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix}, \quad j = 2, \ldots, n-1 \tag{22.2.14}$$

and

$$\mathbf{u}_1 = \mathbf{a}_1 = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \tag{22.2.15}$$

This is a first order recurrence relation for the vectors $\mathbf{u}_j$, and can be solved by the algorithm described above (and implemented in the recipe `recur1`). The only difference is that the multiplications are matrix multiplications with the 2×2 matrices $\mathbf{b}_j$. After the first recursive call, the zeros in **a** and **b** are lost, so we have to write the routine for general two-dimensional vectors and matrices.

Note that this algorithm does not avoid the potential instability problems associated with second order recurrences that are discussed in §5.5 of Volume 1. Also note that the algorithm generalizes in the obvious way to higher-order recurrences: An nth order recurrence can be written as a first order recurrence involving n-dimensional vectors and matrices.

Parallel Solution of Tridiagonal Systems

Closely related to recurrence relations, recursive doubling, and cyclic reduction is the parallel solution of tridiagonal systems. Since Fortran 90 vectors "know their own size," it is most logical to number the components of both the sub- and super-diagonals of the tridiagonal matrix from 1 to $N-1$. Thus equation (2.4.1), here written in the special case of $N = 7$, becomes (blank elements denoting zero),

$$\begin{bmatrix} b_1 & c_1 & & & & & \\ a_1 & b_2 & c_2 & & & & \\ & a_2 & b_3 & c_3 & & & \\ & & a_3 & b_4 & c_4 & & \\ & & & a_4 & b_5 & c_5 & \\ & & & & a_5 & b_6 & c_6 \\ & & & & & a_6 & b_7 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{bmatrix} \tag{22.2.16}$$

The basic idea for solving equation (22.2.16) on a parallel computer is to partition the problem into even and odd elements, recurse to solve the former, and

then solve the latter in parallel. Specifically, we first rewrite (22.2.16), by permuting its rows and columns, as

$$\begin{bmatrix} b_1 & & & & c_1 & & \\ & b_3 & & & a_2 & c_3 & \\ & & b_5 & & & a_4 & c_5 \\ & & & b_7 & & & a_6 \\ a_1 & c_2 & & & b_2 & & \\ & a_3 & c_4 & & & b_4 & \\ & & a_5 & c_6 & & & b_6 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_3 \\ u_5 \\ u_7 \\ u_2 \\ u_4 \\ u_6 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_3 \\ r_5 \\ r_7 \\ r_2 \\ r_4 \\ r_6 \end{bmatrix} \tag{22.2.17}$$

Now observe that, by row operations that subtract multiples of the first four rows from each of the last three rows, we can eliminate all nonzero elements in the lower-left quadrant. The price we pay is bringing some new elements into the lower-right quadrant, whose nonzero elements we now call x's, y's, and z's. We call the modified right-hand sides q. The transformed problem is now

$$\begin{bmatrix} b_1 & & & & c_1 & & \\ & b_3 & & & a_2 & c_3 & \\ & & b_5 & & & a_4 & c_5 \\ & & & b_7 & & & a_6 \\ & & & & y_1 & z_1 & \\ & & & & x_1 & y_2 & z_2 \\ & & & & & x_2 & y_3 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_3 \\ u_5 \\ u_7 \\ u_2 \\ u_4 \\ u_6 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_3 \\ r_5 \\ r_7 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{22.2.18}$$

Notice that the last three rows form a new, smaller, tridiagonal problem, which we can solve simply by recursing! Once its solution is known, the first four rows can be solved by a simple, parallelizable, substitution. This algorithm is implemented in `tridag` in Chapter B2.

The above method is essentially cyclic reduction, but in the case of the tridiagonal problem, it does not "unwind" into a simple iteration; on the contrary, a recursive subroutine is required. For discussion of this and related methods for parallelizing tridiagonal systems, and references to the literature, see Hockney and Jesshope [3].

Recursive doubling can also be used to solve tridiagonal systems, the method requiring the parallel solution (as above) of both a first order recurrence and a second order recurrence [3,4]. For tridiagonal systems, however, cyclic reduction is usually more efficient than recursive doubling.

CITED REFERENCES AND FURTHER READING:

Van Loan, C.F. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.) §1.4.2. [1]

Estrin, G. 1960, quoted in Knuth, D.E. 1981, *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6.4. [2]

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2: Architecture, Programming, and Algorithms* (Bristol and Philadelphia: Adam Hilger), §5.2.4 (cyclic reduction); §5.4.2 (second order recurrences); §5.4 (tridiagonal systems). [3]

Stone, H.S. 1973, *Journal of the ACM*, vol. 20, pp. 27–38; 1975, *ACM Transactions on Mathematical Software*, vol. 1, pp. 289–307. [4]

22.3 Parallel Synthetic Division and Related Algorithms

There are several techniques for parallelization that relate to synthetic division but that can also find application in wider contexts, as we shall see.

Cumulants of a Polynomial

Suppose we have a polynomial

$$P(x) = \sum_{j=0}^{N} c_j x^{N-j} \tag{22.3.1}$$

(Note that, here, the c_j's are indexed from highest degree to lowest, the reverse of the usual convention.) Then we can define the *cumulants* of the polynomial to be partial sums that occur in the polynomial's usual, serial evaluation,

$$\begin{aligned} P_0 &= c_0 \\ P_1 &= c_0 x + c_1 \\ &\cdots \\ P_N &= c_0 x^N + \cdots + c_N = P(x) \end{aligned} \tag{22.3.2}$$

Evidently, the cumulants satisfy a simple, linear first order recurrence relation,

$$P_0 = c_0, \qquad P_j = c_j + x P_{j-1}, \qquad j = 2, \ldots, N \tag{22.3.3}$$

This is slightly simpler than the general first order recurrence, because the value of x does not depend on j. We already know, from §22.2's discussion of recursive doubling, how to parallelize equation (22.3.3) via a recursive subroutine. In Chapter 23, the utility routine `poly_term` will implement just such a procedure. An example of a routine that calls `poly_term` to evaluate a recurrence equivalent to equation (22.3.3) is `eulsum` in Chapter B5.

Notice that while we could use equation (22.3.3), parallelized by recursive doubling, simply to evaluate the polynomial $P(x) = P_N$, this is likely somewhat slower than the alternative technique of vector reduction, also discussed in §22.2, and implemented in the utility function `poly`. Equation (22.3.3) should be saved for cases where the rest of the P_j's (not just P_N) can be put to good use.

Synthetic Division by a Monomial

We now show that evaluation of the cumulants of a polynomial is equivalent to synthetic division of the polynomial by a monomial, also called *deflation* (see §9.5 in Volume 1). To review briefly, and by example, here is a standard tableau from high school algebra for the (long) division of a polynomial $2x^3 - 7x^2 + x + 3$ by the monomial factor $x - 3$.

$$
\begin{array}{r|l}
 & 2x^2 - x - 2 \\
\hline
x-3 & 2x^3 - 7x^2 + x + 3 \\
 & \underline{2x^3 - 6x^2} \\
 & \quad -x^2 + x \\
 & \quad \underline{-x^2 + 3x} \\
 & \qquad -2x + 3 \\
 & \qquad \underline{-2x + 6} \\
 & \qquad\quad -3 \text{ (remainder)}
\end{array}
\tag{22.3.4}
$$

Now, here is the same calculation written as a *synthetic division*, really the same procedure as tableau (22.3.4), but with unnecessary notational baggage omitted (and also a changed sign for the monomial's constant, so that subtractions become additions):

$$
\begin{array}{r|l}
 & \quad 6 \quad -3 \quad -6 \\
\hline
3 & \; 2 \quad -7 \quad +1 \quad +3 \\
\hline
 & \; 2 \quad -1 \quad -2 \quad -3
\end{array}
\tag{22.3.5}
$$

If we substitute symbols for the above quantities with the correspondence

$$
\begin{array}{r|l}
x & \; c_0 \quad c_1 \quad c_2 \quad c_3 \\
\hline
 & \; P_0 \quad P_1 \quad P_2 \quad P_3
\end{array}
\tag{22.3.6}
$$

then it is immediately clear that the P_j's in equation (22.3.6) are simply the P_j's of equation (22.3.3); the calculation is thus parallelizable by recursive doubling. In this context, the utility routine `poly_term` is used by the routine `zroots` in Chapter B9.

Repeated Synthetic Division

It is well known from high-school algebra that repeated synthetic division of a polynomial yields, as the remainders that occur, first the value of the polynomial, next the value of its first derivative, and then (up to multiplication by the factorial of an integer) the values of higher derivatives.

If you want to parallelize the calculation of the value of a polynomial and one or two of its derivatives, it is not unreasonable to evaluate equation (22.3.3), parallelized by recursive doubling, two or three times. Our routine `ddpoly` in Chapter B5 is meant for such use, and it does just this, as does the routine `laguer` in Chapter B9.

There are other cases, however, for which you want to perform repeated synthetic division and "go all the way," until only a constant remains. For example, this is the preferred way of "shifting a polynomial," that is, evaluating the coefficients of a polynomial in a variable y that differs from the original variable x by an additive constant. (The recipe `pcshft` has this as its assigned task.) By way of example, consider the polynomial $3x^3 + x^2 + 4x + 7$, and let us perform repeated synthetic division by a general monomial $x - a$. The conventional calculation then proceeds according to the following tableau, reading it in conventional lexical order (left-to-right and top-to-bottom):

$$
\begin{array}{ccccccc}
3 & & 1 & & 4 & & 7 \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
3 & \xrightarrow{a} & 3a+1 & \xrightarrow{a} & 3a^2+a+4 & \xrightarrow{a} & 3a^3+a^2+4a+7 \\
\downarrow & & \downarrow & & \downarrow & & \\
3 & \xrightarrow{a} & 6a+1 & \xrightarrow{a} & 9a^2+2a+4 & & \\
\downarrow & & \downarrow & & & & \\
3 & \xrightarrow{a} & 9a+1 & & & & \\
\downarrow & & & & & & \\
3 & & & & & &
\end{array}
\tag{22.3.7}
$$

Here, each row (after the first) shows a synthetic division or, equivalently, evaluation of the cumulants of the polynomial whose coefficients are the preceding row. The results at the right edge of the rows are the values of the polynomial and (up to integer factorials) its three nonzero derivatives, or (equivalently, without factorials) coefficients of the shifted polynomial.

We could parallelize the calculation of each row of tableau (22.3.7) by recursive doubling. That is a lot of recursion, which incurs a nonnegligible overhead. A much better way of doing the calculation is to deform tableau (22.3.7) into the following equivalent tableau,

$$
\begin{array}{ccccccc}
3 & \longrightarrow & 3 & & & & \\
 & & \overset{a}{\downarrow} & \searrow & & & \\
1 & \longrightarrow & 3a+1 & & 3 & & \\
 & & \overset{a}{\downarrow} & \searrow & \overset{a}{\downarrow} & \searrow & \\
4 & \longrightarrow & 3a^2+a+4 & & 6a+1 & & 3 \\
 & & \overset{a}{\downarrow} & \searrow & \overset{a}{\downarrow} & \searrow & \overset{a}{\downarrow} \quad \searrow \\
7 & \longrightarrow & 3a^3+a^2+4a+7 & & 9a^2+2a+4 & & 9a+1 \qquad 3
\end{array}
\tag{22.3.8}
$$

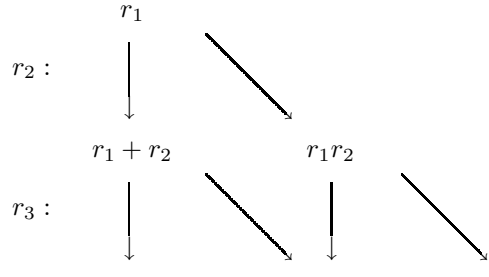
Now each row explicitly depends on only the previous row (and the given first column), so the rows can be calculated in turn by an explicit parallel expression, with no recursive calls needed. An example of coding (22.3.8) in Fortran 90 can be found in the routine `pcshft` in Chapter B5. (It is also possible to eliminate most of the multiplications in (22.3.8), at the expense of a much smaller number of divisions. We have not done this because of the necessity for then treating all possible divisions by zero as special cases. See [1] for details and references.)

Actually, the deformation of (22.3.7) into (22.3.8) is the same trick as was used in Volume 1, p. 167, for evaluating a polynomial and its derivative simultaneously, also generalized in the Fortran 77 implementation of the routine `ddpoly` (Chapter 5). In the Fortran 90 implementation of `ddpoly` (Chapter B5) we *don't* use this trick, but instead use `poly_term`, because, there, we want to parallelize over the length of the polynomial, not over the number of desired derivatives.

Don't confuse the cases of *iterated* synthetic division, discussed here, with the simpler case of doing many simultaneous synthetic divisions. In the latter case, you can simply implement equation (22.3.3) serially, exactly as written, but with each operation being data-parallel across your problem set. (This case occurs in our routine `polcoe` in Chapter B3.)

Polynomial Coefficients from Roots

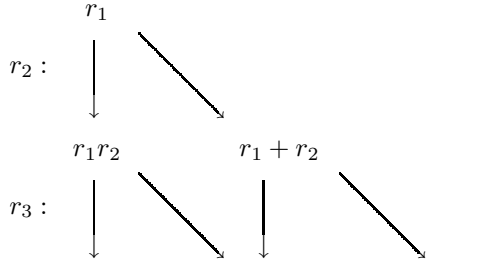
A parallel calculation algorithmically very similar to (22.3.7) or (22.3.8) occurs when we want to find the coefficients of a polynomial $P(x)$ from its roots $r_1, \ldots, r_N$. For this, the tableau is



(22.3.9)

As before, the rows are computed consecutively, from top to bottom. Each row is computed via a single parallel expression. Note that values moving on vertical arrows are simply added in, while values moving on diagonal arrows are multiplied by a new root before adding. Examples of coding (22.3.9) in Fortran 90 can be found in the routines `vander` (Chapter B2) and `polcoe` (Chapter B3).

An equivalent deformation of (22.3.9) is



(22.3.10)

Here the diagonal arrows are simple additions, while the vertical arrows represent multiplication by a root value. Note that the coefficient answers in (22.3.10) come out in the opposite order from (22.3.9). An example of coding (22.3.10) in Fortran 90 can be found in the routine `fixrts` in Chapter B13.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6.4, p. 470. [1]

22.4 Fast Fourier Transforms

Fast Fourier transforms are beloved by computer scientists, especially those who are interested in parallel algorithms, because the FFT's hierarchical structure generates a complicated, but analyzable, set of requirements for interprocessor communication on MMPs. Thus, almost all books on parallel algorithms (e.g., [1–3]) have a chapter on FFTs.

Unfortunately, the resulting algorithms are highly specific to particular parallel architectures, and therefore of little use to us in writing general purpose code in an architecture-independent parallel language like Fortran 90.

Luckily there is a good alternative that covers almost all cases of both serial and parallel machines. If, for a one-dimensional FFT of size N, one is satisfied with parallelism of order $\sqrt{N}$, then there is a good, general way of achieving a parallel FFT with *quite minimal* interprocessor communication; and the communication required is simply the matrix transpose operation, which Fortran 90 implements as an intrinsic. That is the approach that we discuss in this section, and implement in Chapter B12.

For a machine with M processors, this approach will saturate the processors (the desirable condition where none are idle) when the size of a one-dimensional Fourier transform, N, is large enough: $N > M^2$. Smaller N's will not achieve maximum parallelism. But such N's are in fact so small for one-dimensional problems that they are unlikely to be the rate-determining step in scientific calculations. If they are, it is usually because you are doing many such transforms independently, and you should recover "outer parallelism" by doing them all at once.

For two or more dimensions, the adopted approach will saturate M processors when *each* dimension of the problem is larger than M.

Column- and Row-Parallel FFTs

The basic building block that we assume (and implement in Chapter B12) is a routine for simultaneously taking the FFT of each *row* of a two-dimensional matrix. The method is exactly that of Volume 1's `four1` routine, but with array sections like `data(:,j)` replacing scalars like `data(j)`. Chapter B12's implementation of this is a routine called `fourrow`. If all the data for one column (that is, all the values `data(i,:)`, for some `i`) are local to a single processor, then the parallelism involves no interprocessor communication at all: The independent FFTs simply proceed, data parallel and in lockstep. This is architecture-independent parallelism with a vengeance.

We will also need to take the FFT of each *column* of a two-dimensional matrix. One way to do this is to take the transpose (a Fortran 90 intrinsic that hides a lot of interprocessor communication), then take the FFT of the rows using `fourrow`, then take the transpose again. An alternative method is to recode the `four1` routine with array sections in the other dimension (`data(j,:)`) replacing `four1`'s scalars (`data(j)`). This scheme, in Chapter B12, is a routine called `fourcol`. In this case, good parallelism will be achieved only if the values `data(:,i)`, for some `i`, are local to a single processor. Of course, Fortran 90 does not give the user direct control over how data are distributed over the machine; but extensions such as HPF are designed to give just such control.

On a serial machine, you might think that `fourrow` and `fourcol` should have identical timings (acting on a square matrix, say). The two routines do exactly the same operations, after all. Not so! On modern serial computers, `fourrow` and `fourcol` can have timings that differ by a factor of 2 or more, even when their detailed arithmetic is made identical (by giving to one a data array that is the transpose of the data array given to the other). This effect is due to the multilevel cache architecture of most computer memories, and the fact that serial Fortran always stores matrices by columns (first index changing most rapidly). On our workstations, `fourrow` is significantly faster than `fourcol`, and this is likely the generic behavior. However, we do not exclude the possibility that some machines, and some sizes of matrices, are the other way around.

One-Dimensional FFTs

Turn now to the problem of how to do a single, one-dimensional, FFT. We are given a complex array f of length N, an integer power of 2. The basic idea is to address the input array as if it were a two-dimensional array of size $m \times M$, where m and M are each integer powers of 2. Then the components of f can be addressed as

$$f(Jm + j), \qquad 0 \le j < m,\ 0 \le J < M \tag{22.4.1}$$

where the j index changes more rapidly, the J index more slowly, and parentheses denote Fortran-style subscripts.

Now, suppose we had some magical (parallel) method to compute the discrete Fourier transform

$$F(kM + K) \equiv \sum_{j,J} e^{2\pi i(kM+K)(Jm+j)/(Mm)} f(Jm + j),$$
$$0 \le k < m,\ 0 \le K < M \tag{22.4.2}$$

Then, you can see that the indices k and K would address the desired result (FFT of the original array), with K varying more rapidly.

Starting with equation (22.4.2) it is easy to verify the following identity,

$$F(kM + K) = \sum_j \left[e^{2\pi ijk/m} \left(e^{2\pi ijK/(Mm)} \left[\sum_J e^{2\pi iJK/M} f(Jm + j) \right] \right) \right] \tag{22.4.3}$$

But this, reading it from the innermost operation outward, is just the magical method that we need:

- Reshape the original array to $m \times M$ in Fortran normal order (storage by columns).
- FFT on the second (column) index for all values of the first (row) index, using the routine `fourrow`.
- Multiply each component by a phase factor $\exp[2\pi ijK/(Mm)]$.
- Transpose.

- Again FFT on the second (column) index for all values of the first (row) index, using the routine `fourrow`.
- Reshape the two-dimensional array back into one-dimensional output.

The above scheme uses `fourrow` exclusively, on the assumption that it is faster than its sibling `fourcol`. When that is the case (as we typically find), it is likely that the above method, implemented as `four1` in Chapter B12, is faster, even on scalar machines, than Volume 1's scalar version of `four1` (Chapter 12). The reason, as already mentioned, is that `fourrow`'s parallelism is taking better advantage of cache memory locality.

If `fourrow` is *not* faster than `fourcol` on your machine, then you should instead try the following alternative scheme, using `fourcol` only:

- Reshape the original array to $m \times M$ in Fortran normal order (storage by columns).
- Transpose.
- FFT on the first (row) index for all values of the second (column) index, using the routine `fourcol`.
- Multiply each component by a phase factor $\exp[2\pi ijK/(Mm)]$.
- Transpose.
- Again FFT on the first (row) index for all values of the second (column) index, using the routine `fourcol`.
- Transpose.
- Reshape the two-dimensional array back into one-dimensional output.

In Chapter B12, this scheme is implemented as `four1_alt`. You might wonder why `four1_alt` has three transpose operations, while `four1` had only one. Shouldn't there be a symmetry here? No. Fortran makes the arbitrary, but consistent, choice of storing two-dimensional arrays by columns, and this choice favors `four1` in terms of transposes. Luckily, at least on our serial workstations, `fourrow` (used by `four1`) is faster than `fourcol` (used by `four1_alt`), so it is a double win.

For further discussion and references on the ideas behind `four1` and `four1_alt` see [4], where these algorithms are called the four-step and six-step frameworks, respectively.

CITED REFERENCES AND FURTHER READING:

Fox, G.C., et al. 1988, *Solving Problems on Concurrent Processors*, Volume I (Englewood Cliffs, NJ: Prentice Hall), Chapter 11. [1]

Akl, S.G. 1989, *The Design and Analysis of Parallel Algorithms* (Englewood Cliffs, NJ: Prentice Hall), Chapter 9. [2]

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2* (Bristol and Philadelphia: Adam Hilger), §5.5. [3]

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.), §3.3. [4]

22.5 Missing Language Features

A few facilities that are fairly important to parallel programming are missing from the Fortran 90 language standard. On scalar machines this lack is not a

problem, since one can readily program the missing features by using do-loops. On parallel machines, both SSP machines and MMP machines, one must hope that hardware manufacturers provide library routines, callable from Fortran 90, that provide access to the necessary facilities, or use extensions of Fortran 90, such as High Performance Fortran (HPF).

Scatter-with-Combine Functions

Fortran 90 allows the use of *vector subscripts* for so-called *gather* and *scatter* operations. For example, with the setup

```
REAL(SP), DIMENSION(6) :: arr,barr,carr
INTEGER(I4B), DIMENSION(6) :: iarr,jarr
...
iarr = (/ 1,3,5,2,4,6 /)
jarr = (/ 3,2,3,2,1,1 /)
```

Fortran 90 allows both the *one-to-one* gather and the *one-to-many* gather,

```
barr=arr(iarr)
carr=arr(jarr)
```

It also allows the one-to-one scatter,

```
barr(iarr)=carr
```

where the elements of `carr` are "scattered" into `barr` under the direction of the vector subscript `iarr`.

Fortran 90 does *not* allow the *many-to-one* scatter

```
barr(jarr)=carr        ! illegal for this jarr
```

because the repeated values in `jarr` try to assign different components of `carr` to the same location in `barr`. The result would not be deterministic.

Sometimes, however, one would in fact like a many-to-one construction, where the colliding elements get combined by a (commutative and associative) operation, like `+` or `*`, or `max()`. These so-called *scatter-with-combine* functions are readily implemented on serial machines by a do-loop, for example,

```
barr=0.
do j=1,size(carr)
   barr(jarr(j))=barr(jarr(j))+carr(j)
end do
```

Fortran 90 unfortunately provides no means for effecting scatter-with-combine functions in parallel. Luckily, almost all parallel machines do provide such a facility as a library program, as does HPF, where the above facility is called `SUM_SCATTER`. In Chapter 23 we will define utility routines `scatter_add` and `scatter_max` for scatter-with-combine functionalities, but the implementation given in Fortran 90 will be strictly serial, with a do-loop.

Skew Sections

Fortran 90 provides no good, parallel way to access the diagonal elements of a matrix, either to read them or to set them. Do-loops will obviously serve this need on serial machines. In principle, a construction like the following bizarre fragment could also be utilized,

```
REAL(SP), DIMENSION(n,n) :: mat
REAL(SP), DIMENSION(n*n) :: arr
REAL(SP), DIMENSION(n) :: diag
...
arr = reshape(mat,shape(arr))
diag = arr(1:n*n:n+1)
```

which extracts every $(n+1)$st element from a one-dimensional array derived by reshaping the input matrix. However, it is unlikely that any foreseeable parallel compiler will implement the above fragment without a prohibitive amount of unnecessary data movement; and code like the above is also exceedingly slow on all serial machines that we have tried.

In Chapter 23 we will define utility routines `get_diag`, `put_diag`, `diagadd`, `diagmult`, and `unit_matrix` to manipulate diagonal elements, but the implementation given in Fortran 90 will again be strictly serial, with do-loops.

Fortran 95 (see §21.6) will essentially fix Fortran 90's skew sections deficiency. For example, using its `forall` construction, the diagonal elements of an array can be accessed by a statement like

```
forall (j=1:20) diag(j) = arr(j,j)
```

SIMD vs. MIMD

Recall that we use "SIMD" (single-instruction, multiple data) and "data parallel" as interchangeable terms, and that "MIMD" (multiple-instruction, multiple data) is a more general programming model. (See §22.1.)

You should not be too quick to jump to the conclusion that Fortran 90's data parallel or SIMD model is "bad," and that MIMD features, absent in Fortran 90, are therefore "good." On the contrary, Fortran 90's basic data-parallel paradigm has a lot going for it. As we discussed in §22.1, most scientific problems naturally have a "data dimension" across which the time ordering of the calculation is irrelevant. Parallelism across this dimension, which is by nature most often SIMD, frees the mind to think clearly about the computational steps in an algorithm that actually need to be sequential. SIMD code has advantages of clarity and predictability that should not be taken lightly. The general MIMD model of "lots of different things all going on at the same time and communicating data with each other" is a programming and debugging nightmare.

Having said this, we must at the same time admit that a few MIMD features — most notably the ability to go through different logical branches for calculating different data elements in a data-parallel computation — are badly needed in certain programming situations. Fortran 90 is quite weak in this area.

Note that the `where...elsewhere...end where` construction is *not* a MIMD construction. Fortran 90 requires that the `where` clause be executed completely before the `elsewhere` is started. (This allows the results of any calculations in the former

clause to be available for use in the latter.) So, this construction cannot be used to allow two logical branches to be calculated in parallel.

Special functions, where one would like to calculate function values for an array of input quantities, are a particularly compelling example of the need for some MIMD access. Indeed, you will find that Chapter B6 contains a number of intricate, and in a few cases truly bizarre, workarounds, using allowed combinations of `merge`, `where`, and `CONTAINS` (the latter, for separating different logical branches into formally different subprograms).

Fortran 95's `ELEMENTAL` and `PURE` constructions, and to some extent also `forall` (whose body will be able to include `PURE` function calls), will go a long way towards providing exactly the kind of MIMD constructions that are most needed. Once Fortran 95 becomes available and widespread, you can expect to see a new version of this volume, with a much-improved Chapter B6.

Conversely, the number of routines outside of Chapter B6 that can be significantly improved by the use of MIMD features is relatively small; this illustrates the underlying viability of the basic data-parallel SIMD model, even in a future language version with useful MIMD features.

Chapter 23. Numerical Recipes Utility Functions for Fortran 90

23.0 Introduction and Summary Listing

This chapter describes and summarizes the Numerical Recipes utility routines that are used throughout the rest of this volume. A complete implementation of these routines in Fortran 90 is listed in Appendix C1.

Why do we need utility routines? Aren't there already enough of them built into the language as Fortran 90 intrinsics? The answers lie in this volume's dual purpose: to implement the Numerical Recipes routines in Fortran 90 code that runs efficiently on fast serial machines, *and* to implement them, wherever possible, with efficient parallel code for multiprocessor machines that will become increasingly common in the future. We have found three kinds of situations where additional utility routines seem desirable:

1. Fortran 90 is a big language, with many high-level constructs — single statements that actually result in a lot of computing. We like this; it gives the language the potential for expressing algorithms very readably, getting them "out of the mud" of microscopic coding. In coding the 350+ Recipes for this volume, we kept a systematic watch for bits of microscopic coding that were repeated in many routines, and that seemed to be at a lower level of coding than that aspired to by good Fortran 90 style. Once these bits were identified, we pulled them out and substituted calls to new utility routines. These are the utilities that arguably ought to be new language intrinsics, equally useful for serial and parallel machines. (A prime example is `swap`.)

2. Fortran 90 contains many highly parallelizable language constructions. But, as we have seen in §22.5, it is also missing a few important constructions. Most parallel machines will provide these missing elements as machine-coded library subroutines. Some of our utility routines are provided simply as a standard interface to these common, but nonstandard, functionalities. Note that it is the nature of these routines that our specific implementation, in Appendix C1, will be serial, and therefore inefficient on parallel machines. If you have a parallel machine, you will need to recode these; this often involves no more than substituting a one-line library function call for the body of our implementation. Utilities in this category will likely become unnecessary over time, either as machine-dependent libraries converge to standard interfaces, or as the utilities get added to future Fortran

versions. (Indeed, some routines in this category will be unnecessary in Fortran 95, once it is available; see §23.7.)

3. Some tasks should just be done differently in serial, versus parallel, implementation. Linear recurrence relations are a good example (§22.2). These are trivially coded with a do-loop on serial machines, but require a fairly elaborate recursive construction for good parallelization. Rather than provide separate serial and parallel versions of the Numerical Recipes, we have chosen to pull out of the Recipes, and into utility routines, some identifiable tasks of this kind. These are cases where some recoding of our implementation in Appendix C1 might result in improved performance on your particular hardware. Unfortunately, it is not so simple as providing a single "serial implementation" and another single "parallel implementation," because even the seemingly simple word "serial" hides, at the hardware level, a variety of different degrees of pipelining, wide instructions, and so on. Appendix C1 therefore provides only a single implementation, although with some adjustable parameters that you can customize (by experiment) to maximize performance on your hardware.

The above three cases are not really completely distinct, and it is therefore not possible to assign any single utility routine to exactly one situation. Instead, we organize the rest of this chapter as follows: first, an alphabetical list, with short summary, of all the new utility routines; next, a series of short sections, grouped by functionality, that contain the detailed descriptions of the routines.

Alphabetical Listing

The following list gives an abbreviated mnemonic for the type, rank, and/or shape of the returned values (as in §21.4), the routine's calling sequence (optional arguments shown in italics), and a brief, often incomplete, description. The complete description of the routine is given in the later section shown in square brackets.

For each entry, the number shown in parentheses is the approximate number of distinct Recipes in this book that make use of that particular utility function, and is thus a rough guide to that utility's importance. (There may be multiple invocations of the utility in each such Recipe.) Where this number is small or zero, it is usually because the utility routine is a member of a related family of routines whose total usage was deemed significant enough to include, and we did not want users to have to "guess" which family members were instantiated.

`call array_copy(src,dest,n_copied,n_not_copied)`
Copy one-dimensional array (whose size is not necessarily known). [23.1] (9)

[Arr] `arth(first,increment,n)`
Return an arithmetic progression as an array. [23.4] (55)

`call assert(n1,`*`n2,...,string`*`)`
Exit with error message if any logical arguments are false. [23.3] (50)

[Int] `assert_eq(n1,`*`n2,...,string`*`)`
Exit with error message if all integer arguments are not equal; otherwise return common value. [23.3] (133)

[argTS] `cumprod(arr,`*`seed`*`)`

Cumulative products of one-dimensional array, with optional seed value. [23.4] (3)

[argTS] `cumsum(arr,seed)`
Cumulative sums of one-dimensional array, with optional seed value. [23.4] (9)

`call diagadd(mat,diag)`
Adds vector to diagonal of a matrix. [23.7] (4)

`call diagmult(mat,diag)`
Multiplies vector into diagonal of a matrix. [23.7] (2)

[Arr] `geop(first,factor,n)`
Return a geometrical progression as an array. [23.4] (7)

[Arr] `get_diag(mat)`
Gets diagonal of a matrix. [23.7] (2)

[Int] `ifirstloc(arr)`
Location of first true value in a logical array, returned as an integer. [23.2] (3)

[Int] `imaxloc(arr)`
Location of array maximum, returned as an integer. [23.2] (11)

[Int] `iminloc(arr)`
Location of array minimum, returned as an integer. [23.2] (8)

[Mat] `lower_triangle(j,k,extra)`
Returns a lower triangular logical mask. [23.7] (1)

`call nrerror(string)`
Exit with error message. [23.3] (96)

[Mat] `outerand(a,b)`
Returns the outer logical and of two vectors. [23.5] (1)

[Mat] `outerdiff(a,b)`
Returns the outer difference of two vectors. [23.5] (4)

[Mat] `outerdiv(a,b)`
Returns the outer quotient of two vectors. [23.5] (0)

[Mat] `outerprod(a,b)`
Returns the outer product of two vectors. [23.5] (14)

[Mat] `outersum(a,b)`
Returns the outer sum of two vectors. [23.5] (0)

[argTS] `poly(x,coeffs,mask)`
Evaluate a polynomial $P(x)$ for one or more values x, with optional mask. [23.4] (15)

[argTS] `poly_term(a,x)`
Returns partial cumulants of a polynomial, equivalent to synthetic

division. [23.4] (4)

`call put_diag(diag,mat)`
Sets diagonal of a matrix. [23.7] (0)

[Ptr] `reallocate(p,n,`*`m,...`*`)`
Reallocate pointer to new size, preserving its contents. [23.1] (5)

`call scatter_add(dest,source,dest_index)`
Scatter-adds source vector to specified components of destination vector. [23.6] (2)

`call scatter_max(dest,source,dest_index)`
Scatter-max source vector to specified components of destination vector. [23.6] (0)

`call swap(a,b,`*`mask`*`)`
Swap corresponding elements of `a` and `b`. [23.1] (24)

`call unit_matrix(mat)`
Sets matrix to be a unit matrix. [23.7] (6)

[Mat] `upper_triangle(j,k,`*`extra`*`)`
Returns an upper triangular logical mask. [23.7] (4)

[Real] `vabs(v)`
Length of a vector in L_2 norm. [23.8] (6)

[CArr] `zroots_unity(n,nn)`
Returns `nn` consecutive powers of the complex `n`th root of unity. [23.4] (4)

Comment on Relative Frequencies of Use

We find it interesting to compare our frequency of using the `nrutil` utility routines, with our most used language intrinsics (see §21.4). On this basis, the following routines are as useful to us as the *top 10* language intrinsics: `arth`, `assert`, `assert_eq`, `outerprod`, `poly`, and `swap`. We strongly recommend that the X3J3 standards committee, as well as individual compiler library implementors, consider the inclusion of new language intrinsics (or library routines) that subsume the capabilities of at least these routines. In the next tier of importance, we would put some further cumulative operations (`geop`, `cumsum`), some other "outer" operations on vectors (e.g., `outerdiff`), basic operations on the diagonals of matrices (`get_diag`, `put_diag`, `diag_add`), and some means of access to an array of unknown size (`array_copy`).

23.1 Routines That Move Data

To describe our utility routines, we introduce two items of Fortran 90 pseudocode: We use the symbol **T** to denote some type and rank declaration (including

scalar rank, i.e., zero); and when we append a colon to a type specification, as in `INTEGER(I4B)(:)`, for example, we denote an array of the given type.

The routines `swap`, `array_copy`, and `reallocate` simply move data around in useful ways.

⋆ ⋆ ⋆

swap (swaps corresponding elements)

User interface (or, "`USE nrutil`"):

```
SUBROUTINE swap(a,b,mask)
T, INTENT(INOUT) :: a,b
LOGICAL(LGT), INTENT(IN), OPTIONAL :: mask
END SUBROUTINE swap
```

Applicable types and ranks:

T ≡ *any type, any rank*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B)`, `REAL(SP)`, `REAL(SP)(:)`, `REAL(DP)`, `COMPLEX(SPC)`, `COMPLEX(SPC)(:)`, `COMPLEX(SPC)(:,:)`, `COMPLEX(DPC)`, `COMPLEX(DPC)(:)`, `COMPLEX(DPC)(:,:)`

Action:

Swaps the corresponding elements of `a` and `b`. If `mask` is present, performs the swap only where `mask` is true. (Following code is the unmasked case. For speed at run time, the masked case is implemented by overloading, not by testing for the optional argument.)

Reference implementation:

```
T :: dum
dum=a
a=b
b=dum
```

⋆ ⋆ ⋆

array_copy (copy one-dimensional array)

User interface (or, "`USE nrutil`"):

```
SUBROUTINE array_copy(src,dest,n_copied,n_not_copied)
T, INTENT(IN) :: src
T, INTENT(OUT) :: dest
INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
END SUBROUTINE array_copy
```

Applicable types and ranks:

T ≡ *any type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B)(:)`, `REAL(SP)(:)`, `REAL(DP)(:)`

Action:

Copies to a destination array `dest` the one-dimensional array `src`, or as much of `src` as will fit in `dest`. Returns the number of components copied as `n_copied`, and the number of components not copied as `n_not_copied`.

The main use of this utility is where `src` is an expression that returns an array whose size is not known in advance, for example, the value returned by the `pack` intrinsic.

Reference implementation:

```
n_copied=min(size(src),size(dest))
n_not_copied=size(src)-n_copied
dest(1:n_copied)=src(1:n_copied)
```

⋆ ⋆ ⋆

reallocate (reallocate a pointer, preserving contents)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION reallocate(p,n[,m,...])
T, POINTER :: p, reallocate
INTEGER(I4B), INTENT(IN) :: n[,m,...]
END FUNCTION reallocate
```

Applicable types and ranks:

T ≡ *any type, rank 1 or greater*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B)(:), INTEGER(I4B)(:,:), REAL(SP)(:), REAL(SP)(:,:), CHARACTER(1)(:)`

Action:

Allocates storage for a new array with shape specified by the integer(s) `n`, `m`, ... (equal in number to the rank of pointer `p`). Then, copies the contents of `p`'s target (or as much as will fit) into the new storage. Then, deallocates `p` and returns a pointer to the new storage.

The typical use is `p=reallocate(p,n[,m,...])`, which has the effect of changing the allocated size of `p` while preserving the contents.

The reference implementation, below, shows only the case of rank 1.

Reference implementation:

```
INTEGER(I4B) :: nold,ierr
allocate(reallocate(n),stat=ierr)
if (ierr /= 0) call &
   nrerror('reallocate: problem in attempt to allocate memory')
if (.not. associated(p)) RETURN
nold=size(p)
reallocate(1:min(nold,n))=p(1:min(nold,n))
deallocate(p)
```

23.2 Routines Returning a Location

Fortran 90's intrinsics `maxloc` and `minloc` return rank-one arrays. When, in the most frequent usage, their argument is a one-dimensional array, the answer comes back in the inconvenient form of an array containing a single component, which cannot be itself used in a subscript calculation. While there are workaround tricks (e.g., use of the `sum` intrinsic to convert the array to a scalar), it seems clearer to define routines `imaxloc` and `iminloc` that return integers directly.

The routine `ifirstloc` adds a related facility missing among the intrinsics: Return the first true location in a logical array.

⋆ ⋆ ⋆

imaxloc (location of array maximum as an integer)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION imaxloc(arr)
T, INTENT(IN) :: arr
INTEGER(I4B) :: imaxloc
END FUNCTION imaxloc
```

Applicable types and ranks:

T ≡ *any integer or real type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B)(:), REAL(SP)(:)`

Action:

For one-dimensional arrays, identical to the `maxloc` intrinsic, except returns its answer as an integer rather than as `maxloc`'s somewhat awkward rank-one array containing a single component.

Reference implementation:

```
INTEGER(I4B), DIMENSION(1) :: imax
imax=maxloc(arr(:))
imaxloc=imax(1)
```

⋆ ⋆ ⋆

iminloc (location of array minimum as an integer)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION iminloc(arr)
T, INTENT(IN) :: arr
INTEGER(I4B) :: iminloc
END FUNCTION iminloc
```

Applicable types and ranks:

T ≡ *any integer or real type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)(:)`

Action:

For one-dimensional arrays, identical to the `minloc` intrinsic, except returns its answer as an integer rather than as `minloc`'s somewhat awkward rank-one array containing a single component.

Reference implementation:

```
INTEGER(I4B), DIMENSION(1) :: imin
imin=minloc(arr(:))
iminloc=imin(1)
```

⋆ ⋆ ⋆

ifirstloc (returns location of first "true" in a logical vector)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION ifirstloc(mask)
T, INTENT(IN) :: mask
INTEGER(I4B) :: ifirstloc
END FUNCTION ifirstloc
```

Applicable types and ranks:

T ≡ *any logical type, rank 1*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `LOGICAL(LGT)`

Action:

Returns the index (subscript value) of the first location, in a one-dimensional logical `mask`, that has the value `.TRUE.`, or returns `size(mask)+1` if all components of `mask` are `.FALSE.`

Note that while the reference implementation uses a do-loop, the function is parallelized in `nrutil` by instead using the `merge` and `maxloc` intrinsics.

Reference implementation:

```
INTEGER(I4B) :: i
do i=1,size(mask)
   if (mask(i)) then
      ifirstloc=i
      return
   end if
end do
ifirstloc=i
```

23.3 Argument Checking and Error Handling

It is good programming practice for a routine to check the assumptions ("assertions") that it makes about the sizes of input arrays, allowed range of numerical arguments, and so forth. The routines `assert` and `assert_eq` are meant for this kind of use. The routine `nrerror` is our default error reporting routine.

⋆ ⋆ ⋆

assert (exit with error message if any assertion is false)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE assert(n1,n2,...,string)
CHARACTER(LEN=*), INTENT(IN) :: string
LOGICAL, INTENT(IN) :: n1,n2,...
END SUBROUTINE assert
```

Action:

Embedding program dies gracefully with an error message if any of the logical arguments are false. Typical use is with logical expressions as the actual arguments. `nrutil` implements and overloads forms with 1, 2, 3, and 4 logical arguments, plus a form with a vector logical argument,

```
LOGICAL, DIMENSION(:), INTENT(IN) :: n
```

that is checked by the `all(n)` intrinsic.

Reference implementation:

```
if (.not. (n1.and.n2.and...)) then
   write (*,*) 'nrerror: an assertion failed with this tag:', string
   STOP 'program terminated by assert'
end if
```

⋆ ⋆ ⋆

assert_eq (exit with error message if integer arguments not all equal)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION assert_eq(n1,n2,n3,...,string)
CHARACTER(LEN=*), INTENT(IN) :: string
INTEGER, INTENT(IN) :: n1,n2,n3,...
INTEGER :: assert_eq
END FUNCTION assert_eq
```

Action:

Embedding program dies gracefully with an error message if any of the integer arguments are not equal to the first. Otherwise, return the value of the first argument. Typical use is for enforcing equality on the sizes of arrays passed to a subprogram. `nrutil` implements and overloads forms with 1, 2, 3, and 4 integer arguments, plus a form with a vector integer argument,

```
INTEGER, DIMENSION(:), INTENT(IN) :: n
```

that is checked by the conditional `if (all(nn(2:)==nn(1)))`.

Reference implementation:

```
if (n1==n2.and.n2==n3.and...) then
   assert_eq=n1
else
   write (*,*) 'nrerror: an assert_eq failed with this tag:', string
   STOP 'program terminated by assert_eq'
end if
```

⋆ ⋆ ⋆

nrerror (report error message and stop)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE nrerror(string)
CHARACTER(LEN=*), INTENT(IN) :: string
END SUBROUTINE nrerror
```

Action:

This is the minimal error handler used in this book. In applications of any complexity, it is intended only as a placeholder for a user's more complicated error handling strategy.

Reference implementation:

```
write (*,*) 'nrerror: ',string
STOP 'program terminated by nrerror'
```

23.4 Routines for Polynomials and Recurrences

Apart from programming convenience, these routines are designed to allow for nontrivial parallel implementations, as discussed in §22.2 and §22.3.

⋆ ⋆ ⋆

arth (returns arithmetic progression as an array)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION arth(first,increment,n)
T, INTENT(IN) :: first,increment
INTEGER(I4B), INTENT(IN) :: n
T, DIMENSION(n)  [or, 1 rank higher than T]:: arth
END FUNCTION arth
```

Applicable types and ranks:

T ≡ *any numerical type, any rank*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP), REAL(DP)`

Action:

Returns an array of length `n` containing an arithmetic progression whose first value is `first` and whose increment is `increment`. If `first` and `increment` have rank greater than zero, returns an array of one larger rank, with the last subscript having size `n` and indexing the progressions. Note that the following reference implementation (for the scalar case) is definitional only, and neither parallelized nor optimized for roundoff error. See §22.2 and Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
if (n > 0) arth(1)=first
do k=2,n
   arth(k)=arth(k-1)+increment
end do
```

⋆ ⋆ ⋆

geop (returns geometric progression as an array)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION geop(first,factor,n)
T, INTENT(IN) :: first,factor
INTEGER(I4B), INTENT(IN) :: n
T, DIMENSION(n)  [or, 1 rank higher than T]:: geop
END FUNCTION geop
```

Applicable types and ranks:

T ≡ *any numerical type, any rank*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP), REAL(DP), REAL(DP)(:),`
`COMPLEX(SPC)`

Action:

Returns an array of length `n` containing a geometric progression whose first value is `first` and whose multiplier is `factor`. If `first` and `factor` have rank greater than zero, returns an array of one larger rank, with the last subscript having size `n` and indexing the progression. Note that the following reference implementation (for the scalar case) is definitional only, and neither parallelized nor optimized for roundoff error. See §22.2 and Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
if (n > 0) geop(1)=first
do k=2,n
   geop(k)=geop(k-1)*factor
end do
```

⋆ ⋆ ⋆

cumsum (cumulative sum on an array, with optional additive seed)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION cumsum(arr,seed)
T, DIMENSION(:), INTENT(IN) :: arr
T, OPTIONAL, INTENT(IN) :: seed
T, DIMENSION(size(arr)), INTENT(OUT) :: cumsum
END FUNCTION cumsum
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP)`

Action:

Given the rank 1 array `arr` of type **T**, returns an array of identical type and size containing the cumulative sums of `arr`. If the optional argument `seed` is present, it is added to the first component (and therefore, by cumulation, all components) of the result. See §22.2 for parallelization ideas.

Reference implementation:

```
INTEGER(I4B) :: n,j
T :: sd
n=size(arr)
if (n == 0) return
sd=0.0
if (present(seed)) sd=seed
cumsum(1)=arr(1)+sd
do j=2,n
   cumsum(j)=cumsum(j-1)+arr(j)
end do
```

⋆ ⋆ ⋆

cumprod (cumulative prod on an array, with optional multiplicative seed)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION cumprod(arr,seed)
T, DIMENSION(:), INTENT(IN) :: arr
T, OPTIONAL, INTENT(IN) :: seed
T, DIMENSION(size(arr)), INTENT(OUT) :: cumprod
END FUNCTION cumprod
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Given the rank 1 array `arr` of type **T**, returns an array of identical type and size containing the cumulative products of `arr`. If the optional argument `seed` is present, it is multiplied into the first component (and therefore, by cumulation, into all components) of the result. See §22.2 for parallelization ideas.

Reference implementation:

```
INTEGER(I4B) :: n,j
T :: sd
n=size(arr)
if (n == 0) return
sd=1.0
if (present(seed)) sd=seed
cumprod(1)=arr(1)*sd
do j=2,n
   cumprod(j)=cumprod(j-1)*arr(j)
end do
```

⋆ ⋆ ⋆

poly (polynomial evaluation)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION poly(x,coeffs,mask)
T,, DIMENSION(:,...), INTENT(IN) :: x
T, DIMENSION(:), INTENT(IN) :: coeffs
LOGICAL(LGT), DIMENSION(:,...), OPTIONAL, INTENT(IN) :: mask
T :: poly
END FUNCTION poly
```

Applicable types and ranks:

T ≡ *any numerical type (*`x` *may be scalar or have any rank;* `x` *and* `coeffs` *may have different numerical types)*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ *various combinations of* `REAL(SP)`, `REAL(SP)(:)`, `REAL(DP)`, `REAL(DP)(:)`, `COMPLEX(SPC)` *(see Appendix C1 for details)*

Action:

Returns a scalar value or array with the same type and shape as `x`, containing the result of evaluating the polynomial with one-dimensional coefficient vector `coeffs` on each component of `x`. The optional argument `mask`, if present, has the same shape as `x`, and suppresses evaluation of the polynomial where its components are `.false.`. The following reference code shows the case where `mask` is not present. (The other case can be included by overloading.)

Reference implementation:

```
INTEGER(I4B) :: i,n
n=size(coeffs)
if (n <= 0) then
   poly=0.0
else
   poly=coeffs(n)
   do i=n-1,1,-1
      poly=x*poly+coeffs(i)
   end do
end if
```

⋆ ⋆ ⋆

poly_term (partial cumulants of a polynomial)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION poly_term(a,x)
T, DIMENSION(:), INTENT(IN) :: a
T, INTENT(IN) :: x
T, DIMENSION(size(a)) :: poly_term
END FUNCTION poly_term
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP), COMPLEX(SPC)`

Action:

Returns an array of type and size the same as the one-dimensional array `a`, containing the partial cumulants of the polynomial with coefficients `a` (arranged from highest-order to lowest-order coefficients, n.b.) evaluated at `x`. This is equivalent to synthetic division, and can be parallelized. See §22.3. Note that the order of arguments is reversed in `poly` and `poly_term` — each routine returns a value with the size and shape of the *first* argument, the usual Fortran 90 convention.

Reference implementation:

```
INTEGER(I4B) :: n,j
n=size(a)
if (n <= 0) return
poly_term(1)=a(1)
do j=2,n
   poly_term(j)=a(j)+x*poly_term(j-1)
end do
```

⋆ ⋆ ⋆

zroots_unity (returns powers of complex nth root of unity)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION zroots_unity(n,nn)
INTEGER(I4B), INTENT(IN) :: n,nn
COMPLEX(SPC), DIMENSION(nn) :: zroots_unity
END FUNCTION zroots_unity
```

Action:

Returns a complex array containing nn consecutive powers of the nth complex root of unity. Note that the following reference implementation is definitional only, and neither parallelized nor optimized for roundoff error. See Appendix C1 for implementation by subvector scaling.

Reference implementation:

```
INTEGER(I4B) :: k
REAL(SP) :: theta
if (nn==0) return
zroots_unity(1)=1.0
if (nn==1) return
theta=TWOPI/n
zroots_unity(2)=cmplx(cos(theta),sin(theta))
do k=3,nn
   zroots_unity(k)=zroots_unity(k-1)*zroots_unity(2)
end do
```

23.5 Routines for Outer Operations on Vectors

Outer operations on vectors take two vectors as input, and return a matrix as output. One dimension of the matrix is the size of the first vector, the other is the size of the second vector. Our convention is always the standard one,

$$\texttt{result(i,j)} = \texttt{first_operand(i)}\ (op)\ \texttt{second_operand(j)}$$

where (op) is any of addition, subtraction, multiplication, division, and logical and. The reason for coding these as utility routines is that Fortran 90's native construction, with two spreads (cf. §22.1), is difficult to read and thus prone to programmer errors.

⋆ ⋆ ⋆

outerprod (outer product)

User interface (or, "USE nrutil"):

```
FUNCTION outerprod(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outerprod
END FUNCTION outerprod
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in nrutil*:*

T ≡ REAL(SP), REAL(DP)

Action:

Returns a matrix that is the outer product of two vectors.

Reference implementation:

```
outerprod = spread(a,dim=2,ncopies=size(b)) * &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outerdiv (outer quotient)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION outerdiv(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outerdiv
END FUNCTION outerdiv
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Returns a matrix that is the outer quotient of two vectors.

Reference implementation:

```
outerdiv = spread(a,dim=2,ncopies=size(b)) / &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outersum (outer sum)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION outersum(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outersum
END FUNCTION outersum
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Returns a matrix that is the outer sum of two vectors.

Reference implementation:

```
outersum = spread(a,dim=2,ncopies=size(b)) + &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outerdiff (outer difference)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION outerdiff(a,b)
T, DIMENSION(:), INTENT(IN) :: a,b
T, DIMENSION(size(a),size(b)) :: outerdiff
END FUNCTION outerdiff
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `INTEGER(I4B), REAL(SP), REAL(DP)`

Action:

Returns a matrix that is the outer difference of two vectors.

Reference implementation:

```
outerdiff = spread(a,dim=2,ncopies=size(b)) - &
   spread(b,dim=1,ncopies=size(a))
```

⋆ ⋆ ⋆

outerand (outer logical and)

User interface (or, "USE nrutil"):

```
FUNCTION outerand(a,b)
LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: a,b
LOGICAL(LGT), DIMENSION(size(a),size(b)) :: outerand
END FUNCTION outerand
```

Applicable types and ranks:

T ≡ *any logical type*

Types and ranks implemented (overloaded) in nrutil*:*

T ≡ LOGICAL(LGT)

Action:

Returns a matrix that is the outer logical and of two vectors.

Reference implementation:

```
outerand = spread(a,dim=2,ncopies=size(b)) .and. &
   spread(b,dim=1,ncopies=size(a))
```

23.6 Routines for Scatter with Combine

These are common parallel functions that Fortran 90 simply doesn't provide a means for implementing. If you have a parallel machine, you should substitute library routines specific to your hardware.

⋆ ⋆ ⋆

scatter_add (scatter-add source to specified components of destination)

User interface (or, "USE nrutil"):

```
SUBROUTINE scatter_add(dest,source,dest_index)
T, DIMENSION(:), INTENT(OUT) :: dest
T, DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
END SUBROUTINE scatter_add
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in nrutil*:*

T ≡ REAL(SP), REAL(DP)

Action:

Adds each component of the array `source` into a component of `dest` specified by the index array `dest_index`. (The user will usually have zeroed `dest` before the call to this routine.) Note that `dest_index` has the size of `source`, but must contain values in the range from 1 to `size(dest)`, inclusive. Out-of-range values are ignored. There is no parallel implementation of this routine accessible from Fortran 90; most parallel machines supply an implementation as a library routine.

Reference implementation:

```
INTEGER(I4B) :: m,n,j,i
n=assert_eq(size(source),size(dest_index),'scatter_add')
m=size(dest)
do j=1,n
   i=dest_index(j)
   if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
end do
```

⋆ ⋆ ⋆

scatter_max (scatter-max source to specified components of destination)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE scatter_max(dest,source,dest_index)
T, DIMENSION(:), INTENT(OUT) :: dest
T, DIMENSION(:), INTENT(IN) :: source
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
END SUBROUTINE scatter_max
```

Applicable types and ranks:

T ≡ *any integer or real type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP), REAL(DP)`

Action:

Takes the `max` operation between each component of the array `source` and a component of `dest` specified by the index array `dest_index`, replacing that component of `dest` with the value obtained ("maxing into" operation). (The user will often want to fill the array `dest` with the value −`huge` before the call to this routine.) Note that `dest_index` has the size of `source`, but must contain values in the range from 1 to `size(dest)`, inclusive. Out-of-range values are ignored. There is no parallel implementation of this routine accessible from Fortran 90; most parallel machines supply an implementation as a library routine.

Reference implementation:

```
INTEGER(I4B) :: m,n,j,i
n=assert_eq(size(source),size(dest_index),'scatter_max')
m=size(dest)
do j=1,n
   i=dest_index(j)
   if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
end do
```

23.7 Routines for Skew Operations on Matrices

These are also missing parallel capabilities in Fortran 90. In Appendix C1 they are coded serially, with one or more do-loops.

⋆ ⋆ ⋆

diagadd (adds vector to diagonal of a matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE diagadd(mat,diag)
T, DIMENSION(:,:), INTENT(INOUT) :: mat
T, DIMENSION(:), INTENT(IN) :: diag
END SUBROUTINE diagadd
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

The argument `diag`, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix `mat`, is added to the diagonal of the matrix `mat`. The following shows an implementation where `diag` is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

```
INTEGER(I4B) :: j,n
n = assert_eq(size(diag),min(size(mat,1),size(mat,2)),'diagadd')
do j=1,n
   mat(j,j)=mat(j,j)+diag(j)
end do
```

⋆ ⋆ ⋆

diagmult (multiplies vector into diagonal of a matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE diagmult(mat,diag)
T, DIMENSION(:,:), INTENT(INOUT) :: mat
T, DIMENSION(:), INTENT(IN) :: diag
END SUBROUTINE diagmult
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

The argument `diag`, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix `mat`, is multiplied onto the diagonal of the matrix `mat`. The following shows an implementation where `diag` is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

```
INTEGER(I4B) :: j,n
n = assert_eq(size(diag),min(size(mat,1),size(mat,2)),'diagmult')
do j=1,n
   mat(j,j)=mat(j,j)*diag(j)
end do
```

⋆ ⋆ ⋆

get_diag (gets diagonal of matrix)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION get_diag(mat)
T, DIMENSION(:,:), INTENT(IN) :: mat
T, DIMENSION(min(size(mat,1),size(mat,2))) :: get_diag
END FUNCTION get_diag
```

Applicable types and ranks:

T ≡ *any type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP), REAL(DP)`

Action:

Returns a vector containing the diagonal values of the matrix `mat`.

Reference implementation:

```
INTEGER(I4B) :: j
do j=1,min(size(mat,1),size(mat,2))
   get_diag(j)=mat(j,j)
end do
```

⋆ ⋆ ⋆

put_diag (sets the diagonal elements of a matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE put_diag(diag,mat)
T, DIMENSION(:), INTENT(IN) :: diag
T, DIMENSION(:,:), INTENT(INOUT) :: mat
END SUBROUTINE put_diag
```

Applicable types and ranks:

T ≡ *any type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Sets the diagonal of matrix `mat` equal to the argument `diag`, either a scalar or else a vector whose size must be the smaller of the two dimensions of matrix `mat`. The following shows an implementation where `diag` is a vector; the scalar case can be overloaded (see Appendix C1).

Reference implementation:

```
INTEGER(I4B) :: j,n
n=assert_eq(size(diag),min(size(mat,1),size(mat,2)),'put_diag')
do j=1,n
   mat(j,j)=diag(j)
end do
```

⋆ ⋆ ⋆

unit_matrix (returns a unit matrix)

User interface (or, "`USE nrutil`*"):*

```
SUBROUTINE unit_matrix(mat)
T, DIMENSION(:,:), INTENT(OUT) :: mat
END SUBROUTINE unit_matrix
```

Applicable types and ranks:

T ≡ *any numerical type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Sets the diagonal components of `mat` to unity, all other components to zero. When `mat` is square, this will be the unit matrix; otherwise, a unit matrix with appended rows or columns of zeros.

Reference implementation:

```
INTEGER(I4B) :: i,n
n=min(size(mat,1),size(mat,2))
mat(:,:)=0.0
do i=1,n
   mat(i,i)=1.0
end do
```

⋆ ⋆ ⋆

upper_triangle (returns an upper triangular mask)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION upper_triangle(j,k,extra)
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: upper_triangle
END FUNCTION upper_triangle
```

Action:

When the optional argument `extra` is zero or absent, returns a logical mask of shape $(\mathtt{j}, \mathtt{k})$ whose values are true above and to the right of the diagonal, false elsewhere (including on the diagonal). When `extra` is present and positive, a corresponding number of additional (sub-)diagonals are returned as true. ($\mathtt{extra} = 1$ makes the main diagonal return true.) When `extra` is present and negative, it suppresses a corresponding number of superdiagonals.

Reference implementation:

```
INTEGER(I4B) :: n,jj,kk
n=0
if (present(extra)) n=extra
do jj=1,j
   do kk=1,k
      upper_triangle(jj,kk)= (jj-kk < n)
   end do
end do
```

⋆ ⋆ ⋆

lower_triangle (returns a lower triangular mask)

User interface (or, "`USE nrutil`"):

```
FUNCTION lower_triangle(j,k,extra)
INTEGER(I4B), INTENT(IN) :: j,k
INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
LOGICAL(LGT), DIMENSION(j,k) :: lower_triangle
END FUNCTION lower_triangle
```

Action:

When the optional argument `extra` is zero or absent, returns a logical mask of shape $(\mathtt{j}, \mathtt{k})$ whose values are true below and to the left of the diagonal, false elsewhere (including on the diagonal). When `extra` is present and positive, a corresponding number of additional (super-)diagonals are returned as true. ($\mathtt{extra} = 1$ makes the main diagonal return true.) When `extra` is present and negative, it suppresses a corresponding number of subdiagonals.

Reference implementation:

```
INTEGER(I4B) :: n,jj,kk
n=0
if (present(extra)) n=extra
do jj=1,j
   do kk=1,k
      lower_triangle(jj,kk)= (kk-jj < n)
   end do
end do
```

Fortran 95's `forall` construction will make the parallel implementation of all our skew operations utilities extremely simple. For example, the do-loop in `diagadd` will collapse to

```
forall (j=1:n) mat(j,j)=mat(j,j)+diag(j)
```

In fact, this implementation is so simple as to raise the question of whether a separate utility like `diagadd` will be needed at all. There are valid arguments on both sides of this question: The "con" argument, against a routine like `diagadd`, is that it is just another reserved name that you have to remember (if you want to use it). The "pro" argument is that a separate routine avoids the "index pollution" (the opposite disease from "index loss" discussed in §22.1) of introducing a superfluous variable `j`, and that a separate utility allows for additional error checking on the sizes and compatibility of its arguments. We expect that different programmers will have differing tastes.

The argument for keeping a routine like `upper_triangle` or `lower_triangle`, once Fortran 95's *masked* `forall` constructions become available, is less persuasive. We recommend that you consider these two routines as placeholders for "remember to recode this in Fortran 95, someday."

23.8 Other Routine(s)

You might argue that we don't really need a routine for the idiom

```
sqrt(dot_product(v,v))
```

You might be right. The ability to overload the complex case, with its additional complex conjugate, is an argument in its favor, however.

⋆ ⋆ ⋆

vabs (L_2 norm of a vector)

User interface (or, "`USE nrutil`*"):*

```
FUNCTION vabs(v)
T, DIMENSION(:), INTENT(IN) :: v
T :: vabs
END FUNCTION vabs
```

Applicable types and ranks:

T ≡ *any real or complex type*

Types and ranks implemented (overloaded) in `nrutil`*:*

T ≡ `REAL(SP)`

Action:

Returns the length of a vector `v` in L_2 norm, that is, the square root of the sum of the squares of the components. (For complex types, the `dot_product` should be between the vector and its complex conjugate.)

Reference implementation:

```
vabs=sqrt(dot_product(v,v))
```

General Index to Volumes 1 and 2

In this index, page numbers 1 through 934 refer to Volume 1, *Numerical Recipes in Fortran 77*, while page numbers 935 through 1446 refer to Volume 2, *Numerical Recipes in Fortran 90*. Front matter in Volume 1 is indicated by page numbers in the range 1/i through 1/xxxi, while front matter in Volume 2 is indicated 2/i through 2/xx.

Abstract data types 2/xiii, 1030
Accelerated convergence of series 160ff., 1070
Accuracy 19f.
  achievable in minimization 392, 397, 404
  achievable in root finding 346f.
  contrasted with fidelity 832, 840
  CPU different from memory 181
  vs. stability 704, 729, 830, 844
Accuracy parameters 1362f.
Acknowledgments 1/xvi, 2/ix
Ada 2/x
Adams-Bashford-Moulton method 741
Adams' stopping criterion 366
Adaptive integration 123, 135, 703, 708ff., 720, 726, 731f., 737, 742ff., 788, 1298ff., 1303, 1308f.
  Monte Carlo 306ff., 1161ff.
Addition, multiple precision 907, 1353
Addition theorem, elliptic integrals 255
ADI (alternating direction implicit) method 847, 861f., 906
Adjoint operator 867
Adobe Illustrator 1/xvi, 2/xx
Advective equation 826
AGM (arithmetic geometric mean) 906
Airy function 204, 234, 243f.
  routine for 244f., 1121
Aitken's delta squared process 160
Aitken's interpolation algorithm 102
Algol 2/x, 2/xiv
Algorithms, non-numerical 881ff., 1343ff.
Aliasing 495, 569
  see also Fourier transform
all() intrinsic function 945, 948
All-poles model 566
  see also Maximum entropy method (MEM)
All-zeros model 566
  see also Periodogram
Allocatable array 938, 941, 952ff., 1197, 1212, 1266, 1293, 1306, 1336
allocate statement 938f., 941, 953f., 1197, 1266, 1293, 1306, 1336
allocated() intrinsic function 938, 952ff., 1197, 1266, 1293
Allocation status 938, 952ff., 961, 1197, 1266, 1293
Alpha AXP 2/xix
Alternating-direction implicit method (ADI) 847, 861f., 906
Alternating series 160f., 1070
Alternative extended Simpson's rule 128
American National Standards Institute (ANSI) 2/x, 2/xiii
Amoeba 403
  see also Simplex, method of Nelder and Mead
Amplification factor 828, 830, 832, 840, 845f.
Amplitude error 831
Analog-to-digital converter 812, 886
Analyticity 195
Analyze/factorize/operate package 64, 824
Anderson-Darling statistic 621
Andrew's sine 697
Annealing, method of simulated 387f., 436ff., 1219ff.
  assessment 447
  for continuous variables 437, 443ff., 1222
  schedule 438
  thermodynamic analogy 437
  traveling salesman problem 438ff., 1219ff.
ANSI (American National Standards Institute) 2/x, 2/xiii
Antonov-Saleev variant of Sobol' sequence 300, 1160
any() intrinsic function 945, 948
APL (computer language) 2/xi
Apple 1/xxiii
  Macintosh 2/xix, 4, 886
Approximate inverse of matrix 49
Approximation of functions 99, 1043
  by Chebyshev polynomials 185f., 513, 1076ff.
  Padé approximant 194ff., 1080f.
  by rational functions 197ff., 1081f.
  by wavelets 594f., 782
  see also Fitting
Argument
  keyword 2/xiv, 947f., 1341
  optional 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
Argument checking 994f., 1086, 1090, 1092, 1370f.

Arithmetic
  arbitrary precision 881, 906ff., 1352ff.
  floating point 881, 1343
  IEEE standard 276, 882, 1343
  rounding 882, 1343
Arithmetic coding 881, 902ff., 1349ff.
Arithmetic-geometric mean (AGM) method 906
Arithmetic-if statement 2/xi
Arithmetic progression 971f., 996, 1072, 1127, 1365, 1371f.
Array 953ff.
  allocatable 938, 941, 952ff., 1197, 1212, 1266, 1293, 1306, 1336
  allocated with pointer 941
  allocation 953
  array manipulation functions 950
  array sections 939, 941, 943ff.
  of arrays 2/xii, 956, 1336
  associated pointer 953f.
  assumed-shape 942
  automatic 938, 954, 1197, 1212, 1336
  centered subarray of 113
  conformable to a scalar 942f., 965, 1094
  constructor 2/xii, 968, 971, 1022, 1052, 1055, 1127
  copying 991, 1034, 1327f., 1365f.
  cumulative product 997f., 1072, 1086, 1375
  cumulative sum 997, 1280f., 1365, 1375
  deallocation 938, 953f., 1197, 1266, 1293
  disassociated pointer 953
  extents 938, 949
  in Fortran 90 941
  increasing storage for 955, 1070, 1302
  index loss 967f.
  index table 1173ff.
  indices 942
  inquiry functions 948ff.
  intrinsic procedures 2/xiii, 948ff.
  of length 0 944
  of length 1 949
  location of first "true" 993, 1041, 1369
  location of maximum value 993, 1015, 1017, 1365, 1369
  location of minimum value 993, 1369f.
  manipulation functions 950, 1247
  masked swapping of elements in two arrays 1368
  operations on 942, 949, 964ff., 969, 1026, 1040, 1050, 1200, 1326
  outer product 949, 1076
  parallel features 941ff., 964ff., 985
  passing variable number of arguments to function 1022
  of pointers forbidden 956, 1337
  rank 938, 949
  reallocation 955, 992, 1070f., 1365, 1368f.
  reduction functions 948ff.
  shape 938, 944, 949
  size 938
  skew sections 945, 985
  stride 944
  subscript bounds 942
  subscript triplet 944
  swapping elements of two arrays 991, 1015, 1365ff.
  target 938
  three-dimensional, in Fortran 90 1248
  transformational functions 948ff.
  unary and binary functions 949
  undefined status 952ff., 961, 1266, 1293
  zero-length 944
Array section 2/xiii, 943ff., 960
  matches by shape 944
  pointer alias 939, 944f., 1286, 1333
  skew 2/xii, 945, 960, 985, 1284
  vs. eoshift 1078
array_copy() utility function 988, 991, 1034, 1153, 1278, 1328
arth() utility function 972, 974, 988, 996, 1072, 1086, 1127
  replaces do-list 968
Artificial viscosity 831, 837
Ascending transformation, elliptic integrals 256
ASCII character set 6, 888, 896, 902
Assembly language 269
assert() utility function 988, 994, 1086, 1090, 1249
assert_eq() utility function 988, 995, 1022
associated() intrinsic function 952f.
Associated Legendre polynomials 246ff., 764, 1122f., 1319
  recurrence relation for 247
  relation to Legendre polynomials 246
Association, measures of 604, 622ff., 1275
Assumed-shape array 942
Asymptotic series 161
  exponential integral 218
Attenuation factors 583, 1261
Autocorrelation 492
  in linear prediction 558
  use of FFT 538f., 1254
  Wiener-Khinchin theorem 492, 566f.
AUTODIN-II polynomial 890
Automatic array 938, 954, 1197, 1212, 1336
  specifying size of 938, 954
Automatic deallocation 2/xv, 961
Autonomous differential equations 729f.
Autoregressive model (AR) *see* Maximum entropy method (MEM)
Average deviation of distribution 605, 1269
Averaging kernel, in Backus-Gilbert method 807

Backsubstitution 33ff., 39, 42, 92, 1017
  in band diagonal matrix 46, 1021
  in Cholesky decomposition 90, 1039
  complex equations 41
  direct for computing $\mathbf{A}^{-1} \cdot \mathbf{B}$ 40
  with QR decomposition 93, 1040
  relaxation solution of boundary value problems 755, 1316
  in singular value decomposition 56, 1022f.
Backtracking 419
  in quasi-Newton methods 376f., 1195
Backus-Gilbert method 806ff.
Backus, John 2/x
Backward deflation 363

Bader-Deuflhard method 730, 735, 1310f.
Bairstow's method 364, 370, 1193
Balancing 476f., 1230f.
Band diagonal matrix 42ff., 1019
  backsubstitution 46, 1021
  LU decomposition 45, 1020
  multiply by vector 44, 1019
  storage 44, 1019
Band-pass filter 551, 554f.
  wavelets 584, 592f.
Bandwidth limited function 495
Bank accounts, checksum for 894
Bar codes, checksum for 894
Bartlett window 547, 1254ff.
Base case, of recursive procedure 958
Base of representation 19, 882, 1343
BASIC, Numerical Recipes in 1, 2/x, 2/xviii
Basis functions in general linear least squares 665
Bayes' Theorem 810
Bayesian
  approach to inverse problems 799, 810f., 816f.
  contrasted with frequentist 810
  vs. historic maximum entropy method 816f.
  views on straight line fitting 664
Bays' shuffle 270
Bernoulli number 132
Bessel functions 223ff., 234ff., 936, 1101ff.
  asymptotic form 223f., 229f.
  complex 204
  continued fraction 234, 239
  double precision 223
  fractional order 223, 234ff., 1115ff.
  Miller's algorithm 175, 228, 1106
  modified 229ff.
  modified, fractional order 239ff.
  modified, normalization formula 232, 240
  modified, routines for 230ff., 1109ff.
  normalization formula 175
  parallel computation of 1107ff.
  recurrence relation 172, 224, 232, 234
  reflection formulas 236
  reflection formulas, modified functions 241
  routines for 225ff., 236ff., 1101ff.
  routines for modified functions 241ff., 1118
  series for 160, 223
  series for K_ν 241
  series for Y_ν 235
  spherical 234, 245, 1121f.
  turning point 234
  Wronskian 234, 239
Best-fit parameters 650, 656, 660, 698, 1285ff.
  see also Fitting
Beta function 206ff., 1089
  incomplete *see* Incomplete beta function
BFGS algorithm *see* Broyden-Fletcher-Goldfarb-Shanno algorithm
Bias, of exponent 19
Bias, removal in linear prediction 563
Biconjugacy 77
Biconjugate gradient method
  elliptic partial differential equations 824
  preconditioning 78f., 824, 1037
  for sparse system 77, 599, 1034ff.
Bicubic interpolation 118f., 1049f.
Bicubic spline 120f., 1050f.
Big-endian 293
Bilinear interpolation 117
Binary constant, initialization 959
Binomial coefficients 206ff., 1087f.
  recurrences for 209
Binomial probability function 208
  cumulative 222f.
  deviates from 281, 285f., 1155
Binormal distribution 631, 690
Biorthogonality 77
Bisection 111, 359, 1045f.
  compared to minimum bracketing 390ff.
  minimum finding with derivatives 399
  root finding 343, 346f., 352f., 390, 469, 1184f.
BISYNCH 890
Bit 18
  manipulation functions *see* Bitwise logical functions
  reversal in fast Fourier transform (FFT) 499f., 525
bit_size() intrinsic function 951
Bitwise logical functions 2/xiii, 17, 287, 890f., 951
Block-by-block method 788
Block of statements 7
Bode's rule 126
Boltzmann probability distribution 437
Boltzmann's constant 437
Bootstrap method 686f.
Bordering method for Toeplitz matrix 85f.
Borwein and Borwein method for π 906, 1357
Boundary 155f., 425f., 745
Boundary conditions
  for differential equations 701f.
  initial value problems 702
  in multigrid method 868f.
  partial differential equations 508, 819ff., 848ff.
  for spheroidal harmonics 764
  two-point boundary value problems 702, 745ff., 1314ff.
Boundary value problems *see* Differential equations; Elliptic partial differential equations; Two-point boundary value problems
Box-Muller algorithm for normal deviate 279f., 1152
Bracketing
  of function minimum 343, 390ff., 402, 1201f.
  of roots 341, 343ff., 353f., 362, 364, 369, 390, 1183f.
Branch cut, for hypergeometric function 203
Branching 9
Break iteration 14
Brenner, N.M. 500, 517

Brent's method
minimization 389, 395ff., 660f., 1204ff., 1286
minimization, using derivative 389, 399, 1205
root finding 341, 349, 660f., 1188f., 1286
Broadcast (parallel capability) 965ff.
Broyden-Fletcher-Goldfarb-Shanno algorithm 390, 418ff., 1215
Broyden's method 373, 382f., 386, 1199f.
singular Jacobian 386
btest() intrinsic function 951
Bubble sort 321, 1168
Bugs 4
in compilers 1/xvii
how to report 1/iv, 2/iv
Bulirsch-Stoer
algorithm for rational function interpolation 105f., 1043
method (differential equations) 202, 263, 702f., 706, 716, 718ff., 726, 740, 1138, 1303ff.
method (differential equations), stepsize control 719, 726
for second order equations 726, 1307
Burg's LP algorithm 561, 1256
Byte 18

C (programming language) 13, 2/viii
and case construct 1010
Numerical Recipes in 1, 2/x, 2/xvii
C++ 1/xiv, 2/viii, 2/xvi, 7f.
class templates 1083, 1106
Calendar algorithms 1f., 13ff., 1010ff.
Calibration 653
Capital letters in programs 3, 937
Cards, sorting a hand of 321
Carlson's elliptic integrals 255f., 1128ff.
case construct 2/xiv, 1010
trapping errors 1036
Cash-Karp parameters 710, 1299f.
Cauchy probability distribution *see* Lorentzian probability distribution
Cauchy problem for partial differential equations 818f.
Cayley's representation of $\exp(-iHt)$ 844
CCITT (Comité Consultatif International Télégraphique et Téléphonique) 889f., 901
CCITT polynomial 889f.
ceiling() intrinsic function 947
Center of mass 295ff.
Central limit theorem 652f.
Central tendency, measures of 604ff., 1269
Change of variable
in integration 137ff., 788, 1056ff.
in Monte Carlo integration 298
in probability distribution 279
Character functions 952
Character variables, in Fortran 90 1183
Characteristic polynomial
digital filter 554
eigensystems 449, 469
linear prediction 559
matrix with a specified 368, 1193
of recurrence relation 175
Characteristics of partial differential equations 818
Chebyshev acceleration in successive overrelaxation (SOR) 859f., 1332
Chebyshev approximation 84, 124, 183, 184ff., 1076ff.
Clenshaw-Curtis quadrature 190
Clenshaw's recurrence formula 187, 1076
coefficients for 185f., 1076
contrasted with Padé approximation 195
derivative of approximated function 183, 189, 1077f.
economization of series 192f., 195, 1080
for error function 214, 1095
even function 188
and fast cosine transform 513
gamma functions 236, 1118
integral of approximated function 189, 1078
odd function 188
polynomial fits derived from 191, 1078
rational function 197ff., 1081f.
Remes exchange algorithm for filter 553
Chebyshev polynomials 184ff., 1076ff.
continuous orthonormality 184
discrete orthonormality 185
explicit formulas for 184
formula for x^k in terms of 193, 1080
Check digit 894, 1345f.
Checksum 881, 888
cyclic redundancy (CRC) 888ff., 1344f.
Cherry, sundae without a 809
Chi-by-eye 651
Chi-square fitting *see* Fitting; Least squares fitting
Chi-square probability function 209ff., 215, 615, 654, 798, 1272
as boundary of confidence region 688f.
related to incomplete gamma function 215
Chi-square test 614f.
for binned data 614f., 1272
chi-by-eye 651
and confidence limit estimation 688f.
for contingency table 623ff., 1275
degrees of freedom 615f.
for inverse problems 797
least squares fitting 653ff., 1285
nonlinear models 675ff., 1292
rule of thumb 655
for straight line fitting 655ff., 1285
for straight line fitting, errors in both coordinates 660, 1286ff.
for two binned data sets 616, 1272
unequal size samples 617
Chip rate 290
Chirp signal 556
Cholesky decomposition 89f., 423, 455, 1038
backsubstitution 90, 1039
operation count 90
pivoting 90
solution of normal equations 668
Circulant 585
Class, data type 7
Clenshaw-Curtis quadrature 124, 190, 512f.

Clenshaw's recurrence formula 176f., 191, 1078
for Chebyshev polynomials 187, 1076
stability 176f.
Clocking errors 891
CM computers (Thinking Machines Inc.) 964
CM Fortran 2/xv
cn function 261, 1137f.
Coarse-grid correction 864f.
Coarse-to-fine operator 864, 1337
Coding
arithmetic 902ff., 1349ff.
checksums 888, 1344
decoding a Huffman-encoded message 900, 1349
Huffman 896f., 1346ff.
run-length 901
variable length code 896, 1346ff.
Ziv-Lempel 896
see also Arithmetic coding; Huffman coding
Coefficients
binomial 208, 1087f.
for Gaussian quadrature 140ff., 1059ff.
for Gaussian quadrature, nonclassical weight function 151ff., 788f., 1064
for quadrature formulas 125ff., 789, 1328
Cohen, Malcolm 2/xiv
Column degeneracy 22
Column operations on matrix 29, 31f.
Column totals 624
Combinatorial minimization *see* Annealing
Comité Consultatif International Télégraphique et Téléphonique (CCITT) 889f., 901
Common block
obsolescent 2/xif.
superseded by internal subprogram 957, 1067
superseded by module 940, 953, 1298, 1320, 1322, 1324, 1330
Communication costs, in parallel processing 969, 981, 1250
Communication theory, use in adaptive integration 721
Communications protocol 888
Comparison function for rejection method 281
Compilers 964, 1364
CM Fortran 968
DEC (Digital Equipment Corp.) 2/viii
IBM (International Business Machines) 2/viii
Microsoft Fortran PowerStation 2/viii
NAG (Numerical Algorithms Group) 2/viii, 2/xiv
for parallel supercomputers 2/viii
Complementary error function 1094f.
see Error function
Complete elliptic integral *see* Elliptic integrals
Complex arithmetic 171f.
avoidance of in path integration 203
cubic equations 179f.
for linear equations 41
quadratic equations 178
Complex error function 252
Complex plane
fractal structure for Newton's rule 360f.
path integration for function evaluation 201ff., 263, 1138
poles in 105, 160, 202f., 206, 554, 566, 718f.
Complex systems of linear equations 41f.
Compression of data 596f.
Concordant pair for Kendall's tau 637, 1281
Condition number 53, 78
Confidence level 687, 691ff.
Confidence limits
bootstrap method 687f.
and chi-square 688f.
confidence region, confidence interval 687
on estimated model parameters 684ff.
by Monte Carlo simulation 684ff.
from singular value decomposition (SVD) 693f.
Confluent hypergeometric function 204, 239
Conformable arrays 942f., 1094
Conjugate directions 408f., 414ff., 1210
Conjugate gradient method
biconjugate 77, 1034
compared to variable metric method 418
elliptic partial differential equations 824
for minimization 390, 413ff., 804, 815, 1210, 1214
minimum residual method 78
preconditioner 78f., 1037
for sparse system 77ff., 599, 1034
and wavelets 599
Conservative differential equations 726, 1307
Constrained linear inversion method 799ff.
Constrained linear optimization *see* Linear programming
Constrained optimization 387
Constraints, deterministic 804ff.
Constraints, linear 423
CONTAINS statement 954, 957, 1067, 1134, 1202
Contingency coefficient C 625, 1275
Contingency table 622ff., 638, 1275f.
statistics based on chi-square 623ff., 1275
statistics based on entropy 626ff., 1275f.
Continued fraction 163ff.
Bessel functions 234
convergence criterion 165
equivalence transformation 166
evaluation 163ff.
evaluation along with normalization condition 240
even and odd parts 166, 211, 216
even part 249, 251
exponential integral 216
Fresnel integral 248f.
incomplete beta function 219f., 1099f.
incomplete gamma function 211, 1092f.
Lentz's method 165, 212
modified Lentz's method 165
Pincherle's theorem 175
ratio of Bessel functions 239
rational function approximation 164, 211, 219f.
recurrence for evaluating 164f.

and recurrence relation 175
sine and cosine integrals 250f.
Steed's method 164f.
tangent function 164
typography for 163
Continuous variable (statistics) 623
Control structures 7ff., 2/xiv
bad 15
named 959, 1219, 1305
Convergence
accelerated, for series 160ff., 1070
of algorithm for pi 906
criteria for 347, 392, 404, 483, 488, 679, 759
eigenvalues accelerated by shifting 470f.
golden ratio 349, 399
of golden section search 392f.
of Levenberg-Marquardt method 679
linear 346, 393
of QL method 470f.
quadratic 49, 351, 356, 409f., 419, 906
rate 346f., 353, 356
recurrence relation 175
of Ridders' method 351
series vs. continued fraction 163f.
and spectral radius 856ff., 862
Conversion intrinsic functions 946f.
Convex sets, use in inverse problems 804
Convolution
denoted by asterisk 492
finite impulse response (FIR) 531
of functions 492, 503f.
of large data sets 536f.
for multiple precision arithmetic 909, 1354
multiplication as 909, 1354
necessity for optimal filtering 535
overlap-add method 537
overlap-save method 536f.
and polynomial interpolation 113
relation to wavelet transform 585
theorem 492, 531ff., 546
theorem, discrete 531ff.
treatment of end effects 533
use of FFT 523, 531ff., 1253
wraparound problem 533
Cooley-Tukey FFT algorithm 503, 1250
parallel version 1239f.
Co-processor, floating point 886
Copyright rules 1/xx, 2/xix
Cornwell-Evans algorithm 816
Corporate promotion ladder 328
Corrected two-pass algorithm 607, 1269
Correction, in multigrid method 863
Correlation coefficient (linear) 630ff., 1276
Correlation function 492
autocorrelation 492, 539, 558
and Fourier transforms 492
theorem 492, 538
treatment of end effects 538f.
using FFT 538f., 1254
Wiener-Khinchin theorem 492, 566f.
Correlation, statistical 603f., 622
Kendall's tau 634, 637ff., 1279
linear correlation coefficient 630ff., 658, 1276
linear related to least square fitting 630, 658
nonparametric or rank statistical 633ff., 1277
among parameters in a fit 657, 667, 670
in random number generators 268
Spearman rank-order coefficient 634f., 1277
sum squared difference of ranks 634, 1277
Cosine function, recurrence 172
Cosine integral 248, 250ff., 1125f.
continued fraction 250
routine for 251f., 1125
series 250
Cosine transform *see* Fast Fourier transform (FFT); Fourier transform
Coulomb wave function 204, 234
count() intrinsic function 948
Courant condition 829, 832ff., 836
multidimensional 846
Courant-Friedrichs-Lewy stability criterion *see* Courant condition
Covariance
a priori 700
in general linear least squares 667, 671, 1288ff.
matrix, by Cholesky decomposition 91, 667
matrix, of errors 796, 808
matrix, is inverse of Hessian matrix 679
matrix, when it is meaningful 690ff.
in nonlinear models 679, 681, 1292
relation to chi-square 690ff.
from singular value decomposition (SVD) 693f.
in straight line fitting 657
cpu_time() intrinsic function (Fortran 95) 961
CR method *see* Cyclic reduction (CR)
Cramer's V 625, 1275
Crank-Nicholson method 840, 844, 846
Cray computers 964
CRC (cyclic redundancy check) 888ff., 1344f.
CRC-12 890
CRC-16 polynomial 890
CRC-CCITT 890
Creativity, essay on 9
Critical (Nyquist) sampling 494, 543
Cross (denotes matrix outer product) 66
Crosstabulation analysis 623
see also Contingency table
Crout's algorithm 36ff., 45, 1017
cshift() intrinsic function 950
communication bottleneck 969
Cubic equations 178ff., 360
Cubic spline interpolation 107ff., 1044f.
see also Spline
cumprod() utility function 974, 988, 997, 1072, 1086
cumsum() utility function 974, 989, 997, 1280, 1305
Cumulant, of a polynomial 977, 999, 1071f., 1192

Cumulative binomial distribution 222f.
Cumulative Poisson function 214
related to incomplete gamma function 214
Curvature matrix *see* Hessian matrix
cycle statement 959, 1219
Cycle, in multigrid method 865
Cyclic Jacobi method 459, 1225
Cyclic reduction (CR) 848f., 852ff.
linear recurrences 974
tridiagonal systems 976, 1018
Cyclic redundancy check (CRC) 888ff., 1344f.
Cyclic tridiagonal systems 67, 1030

D.C. (direct current) 492
Danielson-Lanczos lemma 498f., 525, 1235ff.
DAP Fortran 2/xi
Data
assigning keys to 889
continuous vs. binned 614
entropy 626ff., 896, 1275
essay on 603
fitting 650ff., 1285ff.
fraudulent 655
glitches in 653
iid (independent and identically distributed) 686
modeling 650ff., 1285ff.
serial port 892
smoothing 604, 644ff., 1283f.
statistical tests 603ff., 1269ff.
unevenly or irregularly sampled 569, 574, 648f., 1258ff.
use of CRCs in manipulating 889
windowing 545ff., 1254
see also Statistical tests
Data compression 596f., 881
arithmetic coding 902ff., 1349ff.
cosine transform 513
Huffman coding 896f., 902, 1346ff.
linear predictive coding (LPC) 563ff.
lossless 896
Data Encryption Standard (DES) 290ff., 1144, 1147f., 1156ff.
Data hiding 956ff., 1209, 1293, 1296
Data parallelism 941, 964ff., 985
DATA statement 959
for binary, octal, hexadecimal constants 959
repeat count feature 959
superseded by initialization expression 943, 959, 1127
Data type 18, 936
accuracy parameters 1362f.
character 1183
derived 2/xiii, 937, 1030, 1336, 1346
derived, for array of arrays 956, 1336
derived, initialization 2/xv
derived, for Numerical Recipes 1361
derived, storage allocation 955
DP (double precision) 1361f.
DPC (double precision complex) 1361
I1B (1 byte integer) 1361
I2B (2 byte integer) 1361
I4B (4 byte integer) 1361
intrinsic 937
LGT (default logical type) 1361
nrtype.f90 1361f.
passing complex as real 1140
SP (single precision) 1361f.
SPC (single precision complex) 1361
user-defined 1346
DAUB4 584ff., 588, 590f., 594, 1264f.
DAUB6 586
DAUB12 598
DAUB20 590f., 1265
Daubechies wavelet coefficients 584ff., 588, 590f., 594, 598, 1264ff.
Davidon-Fletcher-Powell algorithm 390, 418ff., 1215
Dawson's integral 252ff., 600, 1127f.
approximation for 252f.
routine for 253f., 1127
dble() intrinsic function (deprecated) 947
deallocate statement 938f., 953f., 1197, 1266, 1293
Deallocation, of allocatable array 938, 953f., 1197, 1266, 1293
Debugging 8
DEC (Digital Equipment Corp.) 1/xxiii, 2/xix, 886
Alpha AXP 2/viii
Fortran 90 compiler 2/viii
quadruple precision option 1362
VAX 4
Decomposition *see* Cholesky decomposition; LU decomposition; QR decomposition; Singular value decomposition (SVD)
Deconvolution 535, 540, 1253
see also Convolution; Fast Fourier transform (FFT); Fourier transform
Defect, in multigrid method 863
Deferred approach to the limit *see* Richardson's deferred approach to the limit
Deflation
of matrix 471
of polynomials 362ff., 370f., 977
Degeneracy of linear algebraic equations 22, 53, 57, 670
Degenerate kernel 785
Degenerate minimization principle 795
Degrees of freedom 615f., 654, 691
Dekker, T.J. 353
Demonstration programs 3, 936
Deprecated features
common block 2/xif., 940, 953, 957, 1067, 1298, 1320, 1322, 1324, 1330
dble() intrinsic function 947
EQUIVALENCE statement 2/xif., 1161, 1286
statement function 1057, 1256
Derivatives
computation via Chebyshev approximation 183, 189, 1077f.
computation via Savitzky-Golay filters 183, 645
matrix of first partial *see* Jacobian determinant
matrix of second partial *see* Hessian matrix

numerical computation 180ff., 379, 645, 732, 750, 771, 1075, 1197, 1309
of polynomial 167, 978, 1071f.
use in optimization 388f., 399, 1205ff.
Derived data type *see* Data type, derived
DES *see* Data Encryption Standard
Descending transformation, elliptic integrals 256
Descent direction 376, 382, 419
Descriptive statistics 603ff., 1269ff.
see also Statistical tests
Design matrix 645, 665, 795, 801, 1082
Determinant 25, 41
Deviates, random *see* Random deviates
DFP algorithm *see* Davidon-Fletcher-Powell algorithm
diagadd() utility function 985, 989, 1004
diagmult() utility function 985, 989, 1004, 1294
Diagonal dominance 43, 679, 780, 856
Difference equations, finite *see* Finite difference equations (FDEs)
Difference operator 161
Differential equations 701ff., 1297ff.
accuracy vs. stability 704, 729
Adams-Bashforth-Moulton schemes 741
adaptive stepsize control 703, 708ff., 719, 726, 731, 737, 742f., 1298ff., 1303ff., 1308f., 1311ff.
algebraically difficult sets 763
backward Euler's method 729
Bader-Deuflhard method for stiff 730, 735, 1310f.
boundary conditions 701f., 745ff., 749, 751f., 771, 1314ff.
Bulirsch-Stoer method 202, 263, 702, 706, 716, 718ff., 740, 1138, 1303
Bulirsch-Stoer method for conservative equations 726, 1307
comparison of methods 702f., 739f., 743
conservative 726, 1307
danger of too small stepsize 714
eigenvalue problem 748, 764ff., 770ff., 1319ff.
embedded Runge-Kutta method 709f., 731, 1298, 1308
equivalence of multistep and multivalue methods 743
Euler's method 702, 704, 728f.
forward Euler's method 728
free boundary problem 748, 776
high-order implicit methods 730ff., 1308ff.
implicit differencing 729, 740, 1308
initial value problems 702
internal boundary conditions 775ff.
internal singular points 775ff.
interpolation on right-hand sides 111
Kaps-Rentrop method for stiff 730, 1308
local extrapolation 709
modified midpoint method 716f., 719, 1302f.
multistep methods 740ff.
multivalue methods 740
order of method 704f., 719
path integration for function evaluation 201ff., 263, 1138
predictor-corrector methods 702, 730, 740ff.
reduction to first-order sets 701, 745
relaxation method 746f., 753ff., 1316ff.
relaxation method, example of 764ff., 1319ff.
r.h.s. independent of x 729f.
Rosenbrock methods for stiff 730, 1308f.
Runge-Kutta method 702, 704ff., 708ff., 731, 740, 1297f., 1308
Runge-Kutta method, high-order 705, 1297
Runge-Kutta-Fehlberg method 709ff., 1298
scaling stepsize to required accuracy 709
second order 726, 1307
semi-implicit differencing 730
semi-implicit Euler method 730, 735f.
semi-implicit extrapolation method 730, 735f., 1311ff.
semi-implicit midpoint rule 735f., 1310f.
shooting method 746, 749ff., 1314ff.
shooting method, example 770ff., 1321ff.
similarity to Volterra integral equations 786
singular points 718f., 751, 775ff., 1315f., 1323ff.
step doubling 708f.
stepsize control 703, 708ff., 719, 726, 731, 737, 742f., 1298, 1303ff., 1308f.
stiff 703, 727ff., 1308ff.
stiff methods compared 739
Stoermer's rule 726, 1307
see also Partial differential equations; Two-point boundary value problems
Diffusion equation 818, 838ff., 855
Crank-Nicholson method 840, 844, 846
Forward Time Centered Space (FTCS) 839ff., 855
implicit differencing 840
multidimensional 846
Digamma function 216
Digital filtering *see* Filter
Dihedral group D_5 894
dim optional argument 948
Dimensional expansion 965ff.
Dimensions (units) 678
Diminishing increment sort 322, 1168
Dirac delta function 284, 780
Direct method *see* Periodogram
Direct methods for linear algebraic equations 26, 1014
Direct product *see* Outer product of matrices
Direction of largest decrease 410f.
Direction numbers, Sobol's sequence 300
Direction-set methods for minimization 389, 406f., 1210ff.
Dirichlet boundary conditions 820, 840, 850, 856, 858
Disclaimer of warranty 1/xx, 2/xvii
Discordant pair for Kendall's tau 637, 1281
Discrete convolution theorem 531ff.

Discrete Fourier transform (DFT) 495ff., 1235ff.
as approximate continuous transform 497
see also Fast Fourier transform (FFT)
Discrete optimization 436ff., 1219ff.
Discriminant 178, 457
Diskettes
are ANSI standard 3
how to order 1/xxi, 2/xvii
Dispersion 831
DISPO *see* Savitzky-Golay filters
Dissipation, numerical 830
Divergent series 161
Divide and conquer algorithm 1226, 1229
Division
complex 171
multiple precision 910f., 1356
of polynomials 169, 362, 370, 1072
dn function 261, 1137f.
Do-list, implied 968, 971, 1127
Do-loop 2/xiv
Do-until iteration 14
Do-while iteration 13
Dogleg step methods 386
Domain of integration 155f.
Dominant solution of recurrence relation 174
Dot (denotes matrix multiplication) 23
dot_product() intrinsic function 945, 949, 969, 1216
Double exponential error distribution 696
Double precision
converting to 1362
as refuge of scoundrels 882
use in iterative improvement 47, 1022
Double root 341
Downhill simplex method *see* Simplex, method of Nelder and Mead
DP, defined 937
Driver programs 3
Dual viewpoint, in multigrid method 875
Duplication theorem, elliptic integrals 256
DWT (discrete wavelet transform) *see* Wavelet transform
Dynamical allocation of storage 2/xiii, 869, 938, 941f., 953ff., 1327, 1336
garbage collection 956
increasing 955, 1070, 1302

Eardley, D.M. 338
EBCDIC 890
Economization of power series 192f., 195, 1080
Eigensystems 449ff., 1225ff.
balancing matrix 476f., 1230f.
bounds on eigenvalues 50
calculation of few eigenvalues 454, 488
canned routines 454f.
characteristic polynomial 449, 469
completeness 450
defective 450, 476, 489
deflation 471
degenerate eigenvalues 449ff.
elimination method 453, 478, 1231
factorization method 453
fast Givens reduction 463
generalized eigenproblem 455
Givens reduction 462f.
Hermitian matrix 475
Hessenberg matrix 453, 470, 476ff., 488, 1232
Householder transformation 453, 462ff., 469, 473, 475, 478, 1227f., 1231
ill-conditioned eigenvalues 477
implicit shifts 472ff., 1228f.
and integral equations 779, 785
invariance under similarity transform 452
inverse iteration 455, 469, 476, 487ff., 1230
Jacobi transformation 453, 456ff., 462, 475, 489, 1225f.
left eigenvalues 451
list of tasks 454f.
multiple eigenvalues 489
nonlinear 455
nonsymmetric matrix 476ff., 1230ff.
operation count of balancing 476
operation count of Givens reduction 463
operation count of Householder reduction 467
operation count of inverse iteration 488
operation count of Jacobi method 460
operation count of QL method 470, 473
operation count of QR method for Hessenberg matrices 484
operation count of reduction to Hessenberg form 479
orthogonality 450
parallel algorithms 1226, 1229
polynomial roots and 368, 1193
QL method 469ff., 475, 488f.
QL method with implicit shifts 472ff., 1228f.
QR method 52, 453, 456, 469ff., 1228
QR method for Hessenberg matrices 480ff., 1232ff.
real, symmetric matrix 150, 467, 785, 1225, 1228
reduction to Hessenberg form 478f., 1231
right eigenvalues 451
shifting eigenvalues 449, 470f., 480
special matrices 454
termination criterion 484, 488
tridiagonal matrix 453, 469ff., 488, 1228
Eigenvalue and eigenvector, defined 449
Eigenvalue problem for differential equations 748, 764ff., 770ff., 1319ff.
Eigenvalues and polynomial root finding 368, 1193
EISPACK 454, 475
Electromagnetic potential 519
ELEMENTAL attribute (Fortran 95) 961, 1084
Elemental functions 2/xiii, 2/xv, 940, 942, 946f., 961, 986, 1015, 1083, 1097f.
Elimination *see* Gaussian elimination
Ellipse in confidence limit estimation 688
Elliptic integrals 254ff., 906
addition theorem 255

Carlson's forms and algorithms 255f., 1128ff.
Cauchy principal value 256f.
duplication theorem 256
Legendre 254ff., 260f., 1135ff.
routines for 257ff., 1128ff.
symmetric form 255
Weierstrass 255
Elliptic partial differential equations 818, 1332ff.
alternating-direction implicit method (ADI) 861f., 906
analyze/factorize/operate package 824
biconjugate gradient method 824
boundary conditions 820
comparison of rapid methods 854
conjugate gradient method 824
cyclic reduction 848f., 852ff.
Fourier analysis and cyclic reduction (FACR) 848ff., 854
Gauss-Seidel method 855, 864ff., 876, 1338, 1341
incomplete Cholesky conjugate gradient method (ICCG) 824
Jacobi's method 855f., 864
matrix methods 824
multigrid method 824, 862ff., 1009, 1334ff.
rapid (Fourier) method 824, 848ff.
relaxation method 823, 854ff., 1332
strongly implicit procedure 824
successive over-relaxation (SOR) 857ff., 862, 866, 1332
elsewhere construct 943
Emacs, GNU 1/xvi
Embedded Runge-Kutta method 709f., 731, 1298, 1308
Encapsulation, in programs 7
Encryption 290, 1156
enddo statement 12, 17
Entropy 896
of data 626ff., 811, 1275
EOM (end of message) 902
eoshift() intrinsic function 950
communication bottleneck 969
vector shift argument 1019f.
vs. array section 1078
epsilon() intrinsic function 951, 1189
Equality constraints 423
Equations
cubic 178ff., 360
normal (fitting) 645, 666ff., 800, 1288
quadratic 20, 178
see also Differential equations; Partial differential equations; Root finding
Equivalence classes 337f., 1180
EQUIVALENCE statement 2/xif., 1161, 1286
Equivalence transformation 166
Error
checksums for preventing 891
clocking 891
double exponential distribution 696
local truncation 875
Lorentzian distribution 696f.
in multigrid method 863
nonnormal 653, 690, 694ff.
relative truncation 875
roundoff 180f., 881, 1362
series, advantage of an even 132f., 717, 1362
systematic vs. statistical 653, 1362
truncation 20f., 180, 399, 709, 881, 1362
varieties found by check digits 895
varieties of, in PDEs 831ff.
see also Roundoff error
Error function 213f., 601, 1094f.
approximation via sampling theorem 601
Chebyshev approximation 214, 1095
complex 252
for Fisher's z-transformation 632, 1276
relation to Dawson's integral 252, 1127
relation to Fresnel integrals 248
relation to incomplete gamma function 213
routine for 214, 1094
for significance of correlation 631, 1276
for sum squared difference of ranks 635, 1277
Error handling in programs 2/xii, 2/xvi, 3, 994f., 1036, 1370f.
Estimation of parameters *see* Fitting; Maximum likelihood estimate
Estimation of power spectrum 542ff., 565ff., 1254ff., 1258
Euler equation (fluid flow) 831
Euler-Maclaurin summation formula 132, 135
Euler's constant 216ff., 250
Euler's method for differential equations 702, 704, 728f.
Euler's transformation 160f., 1070
generalized form 162f.
Evaluation of functions *see* Function
Even and odd parts, of continued fraction 166, 211, 216
Even parity 888
Exception handling in programs *see* Error handling in programs
exit statement 959, 1219
Explicit differencing 827
Exponent in floating point format 19, 882, 1343
exponent intrinsic function 1107
Exponential deviate 278, 1151f.
Exponential integral 215ff., 1096f.
asymptotic expansion 218
continued fraction 216
recurrence relation 172
related to incomplete gamma function 215
relation to cosine integral 250
routine for $\mathrm{Ei}(x)$ 218, 1097
routine for $E_n(x)$ 217, 1096
series 216
Exponential probability distribution 570
Extended midpoint rule 124f., 129f., 135, 1054f.
Extended Simpson's rule 128, 788, 790
Extended Simpson's three-eighths rule 789
Extended trapezoidal rule 125, 127, 130ff., 135, 786, 1052ff., 1326
roundoff error 132
Extirpolation (so-called) 574, 1261

Extrapolation 99ff.
in Bulirsch-Stoer method 718ff., 726, 1305ff.
differential equations 702
by linear prediction 557ff., 1256f.
local 709
maximum entropy method as type of 567
polynomial 724, 726, 740, 1305f.
rational function 718ff., 726, 1306f.
relation to interpolation 101
for Romberg integration 134
see also Interpolation
Extremization *see* Minimization

F-distribution probability function 222
F-test for differences of variances 611, 613, 1271
FACR *see* Fourier analysis and cyclic reduction (FACR)
Facsimile standard 901
Factorial
double (denoted "!!") 247
evaluation of 159, 1072, 1086
relation to gamma function 206
routine for 207f., 1086ff.
False position 347ff., 1185f.
Family tree 338
FAS (full approximation storage algorithm) 874, 1339ff.
Fast Fourier transform (FFT) 498ff., 881, 981, 1235f.
alternative algorithms 503f.
as approximation to continuous transform 497
Bartlett window 547, 1254
bit reversal 499f., 525
and Clenshaw-Curtis quadrature 190
column-parallel algorithm 981, 1237ff.
communication bottleneck 969, 981, 1250
convolution 503f., 523, 531ff., 909, 1253, 1354
convolution of large data sets 536f.
Cooley-Tukey algorithm 503, 1250
Cooley-Tukey algorithm, parallel 1239f.
correlation 538f., 1254
cosine transform 190, 511ff., 851, 1245f.
cosine transform, second form 513, 852, 1246
Danielson-Lanczos lemma 498f., 525
data sets not a power of 2 503
data smoothing 645
data windowing 545ff., 1254
decimation-in-frequency algorithm 503
decimation-in-time algorithm 503
discrete autocorrelation 539, 1254
discrete convolution theorem 531ff.
discrete correlation theorem 538
at double frequency 575
effect of caching 982
endpoint corrections 578f., 1261ff.
external storage 525
figures of merit for data windows 548
filtering 551ff.
FIR filter 553
four-step framework 983, 1239
Fourier integrals 577ff., 1261
Fourier integrals, infinite range 583
Hamming window 547
Hann window 547
history 498
IIR filter 553ff.
image processing 803, 805
integrals using 124
inverse of cosine transform 512ff.
inverse of sine transform 511
large data sets 525
leakage 544
memory-local algorithm 528
multidimensional 515ff., 1236f., 1241, 1246, 1251
for multiple precision arithmetic 906
for multiple precision multiplication 909, 1354
number-theoretic transforms 503f.
operation count 498
optimal (Wiener) filtering 539ff., 558
order of storage in 501
parallel algorithms 981ff., 1235ff.
partial differential equations 824, 848ff.
Parzen window 547
periodicity of 497
periodogram 543ff., 566
power spectrum estimation 542ff., 1254ff.
for quadrature 124
of real data in 2D and 3D 519ff., 1248f.
of real functions 504ff., 519ff., 1242f., 1248f.
related algorithms 503f.
row-parallel algorithm 981, 1235f.
Sande-Tukey algorithm 503
sine transform 508ff., 850, 1245
Singleton's algorithm 525
six-step framework 983, 1240
square window 546, 1254
timing 982
treatment of end effects in convolution 533
treatment of end effects in correlation 538f.
Tukey's trick for frequency doubling 575
use in smoothing data 645
used for Lomb periodogram 574, 1259
variance of power spectrum estimate 544f., 549
virtual memory machine 528
Welch window 547, 1254
Winograd algorithms 503
see also Discrete Fourier transform (DFT); Fourier transform; Spectral density
Faure sequence 300
Fax (facsimile) Group 3 standard 901
Feasible vector 424
FFT *see* Fast Fourier transform (FFT)
Field, in data record 329
Figure-of-merit function 650
Filon's method 583
Filter 551ff.
acausal 552
bilinear transformation method 554
causal 552, 644

characteristic polynomial 554
data smoothing 644f., 1283f.
digital 551ff.
DISPO 644
by fast Fourier transform (FFT) 523, 551ff.
finite impulse response (FIR) 531, 552
homogeneous modes of 554
infinite impulse response (IIR) 552ff., 566
Kalman 700
linear 552ff.
low-pass for smoothing 644ff., 1283f.
nonrecursive 552
optimal (Wiener) 535, 539ff., 558, 644
quadrature mirror 585, 593
realizable 552, 554f.
recursive 552ff., 566
Remes exchange algorithm 553
Savitzky-Golay 183, 644ff., 1283f.
stability of 554f.
in the time domain 551ff.
Fine-to-coarse operator 864, 1337
Finite difference equations (FDEs) 753, 763, 774
alternating-direction implicit method (ADI) 847, 861f.
art not science 829
Cayley's form for unitary operator 844
Courant condition 829, 832ff., 836
Courant condition (multidimensional) 846
Crank-Nicholson method 840, 844, 846
eigenmodes of 827f.
explicit vs. implicit schemes 827
forward Euler 826f.
Forward Time Centered Space (FTCS) 827ff., 839ff., 843, 855
implicit scheme 840
Lax method 828ff., 836
Lax method (multidimensional) 845f.
mesh drifting instability 834f.
numerical derivatives 181
partial differential equations 821ff.
in relaxation methods 753ff.
staggered leapfrog method 833f.
two-step Lax-Wendroff method 835ff.
upwind differencing 832f., 837
see also Partial differential equations
Finite element methods, partial differential equations 824
Finite impulse response (FIR) 531
Finkelstein, S. 1/xvi, 2/ix
FIR (finite impulse response) filter 552
Fisher's z-transformation 631f., 1276
Fitting 650ff., 1285ff.
basis functions 665
by Chebyshev approximation 185f., 1076
chi-square 653ff., 1285ff.
confidence levels related to chi-square values 691ff.
confidence levels from singular value decomposition (SVD) 693f.
confidence limits on fitted parameters 684ff.
covariance matrix not always meaningful 651, 690
degeneracy of parameters 674
an exponential 674
freezing parameters in 668, 700
Gaussians, a sum of 682, 1294
general linear least squares 665ff., 1288, 1290f.
Kalman filter 700
K–S test, caution regarding 621f.
least squares 651ff., 1285
Legendre polynomials 674, 1291f.
Levenberg-Marquardt method 678ff., 816, 1292f.
linear regression 655ff., 1285ff.
maximum likelihood estimation 652f., 694ff.
Monte Carlo simulation 622, 654, 684ff.
multidimensional 675
nonlinear models 675ff., 1292f.
nonlinear models, advanced methods 683
nonlinear problems that are linear 674
nonnormal errors 656, 690, 694ff.
polynomial 83, 114, 191, 645, 665, 674, 1078, 1291
by rational Chebyshev approximation 197ff., 1081f.
robust methods 694ff., 1294
of sharp spectral features 566
standard (probable) errors on fitted parameters 657f., 661, 667, 671, 684ff., 1285f., 1288, 1290
straight line 655ff., 667f., 698, 1285ff., 1294ff.
straight line, errors in both coordinates 660ff., 1286ff.
see also Error; Least squares fitting; Maximum likelihood estimate; Robust estimation
Five-point difference star 867
Fixed point format 18
Fletcher-Powell algorithm *see* Davidon-Fletcher-Powell algorithm
Fletcher-Reeves algorithm 390, 414ff., 1214
Floating point co-processor 886
Floating point format 18ff., 882, 1343
care in numerical derivatives 181
IEEE 276, 882, 1343
floor() intrinsic function 948
Flux-conservative initial value problems 825ff.
FMG (full multigrid method) 863, 868, 1334ff.
FOR iteration 9f., 12
forall statement 2/xii, 2/xv, 960, 964, 986
access to associated index 968
skew array sections 985, 1007
Formats of numbers 18ff., 882, 1343
Fortran 9
arithmetic-if statement 2/xi
COMMON block 2/xif., 953, 957
deprecated features 2/xif., 947, 1057, 1161, 1256, 1286
dynamical allocation of storage 869, 1336
EQUIVALENCE statement 2/xif., 1161, 1286
evolution of 2/xivff.
exception handling 2/xii, 2/xvi
filenames 935
Fortran 2000 (planned) 2/xvi

Fortran 95 2/xv, 945, 947, 1084, 1100, 1364
HPF (High-Performance Fortran) 2/xvf.
Numerical Recipes in 2/x, 2/xvii, 1
obsolescent features 2/xif.
side effects 960
see also Fortran 90
Fortran D 2/xv
Fortran 77 1/xix
bit manipulation functions 17
hexadecimal constants 17
Fortran 8x 2/xi, 2/xiii
Fortran 90 3
abstract data types 2/xiii, 1030
all() intrinsic function 945, 948
allocatable array 938, 941, 953ff., 1197, 1212, 1266, 1293, 1306, 1336
allocate statement 938f., 941, 953f., 1197, 1266, 1293, 1306, 1336
allocated() intrinsic function 938, 952ff., 1197, 1266, 1293
any() intrinsic function 945, 948
array allocation and deallocation 953
array of arrays 2/xii, 956, 1336
array constructor 2/xii, 968, 971, 1022, 1052, 1055, 1127
array constructor with implied do-list 968, 971
array extents 938, 949
array features 941ff., 953ff.
array intrinsic procedures 2/xiii, 948ff.
array of length 0 944
array of length 1 949
array manipulation functions 950
array parallel operations 964f.
array rank 938, 949
array reallocation 955
array section 2/xiif., 2/xiii, 939, 941ff., 960, 1078, 1284, 1286, 1333
array shape 938, 949
array size 938, 942
array transpose 981f.
array unary and binary functions 949
associated() intrinsic function 952f.
associated pointer 953f.
assumed-shape array 942
automatic array 938, 954, 1197, 1212, 1336
backwards-compatibility 935, 946
bit manipulation functions 2/xiii, 951
bit_size() intrinsic function 951
broadcasts 965f.
btest() intrinsic function 951
case construct 1010, 1036
case insensitive 937
ceiling() intrinsic function 947
character functions 952
character variables 1183
cmplx function 1125
communication bottlenecks 969, 981, 1250
compatibility with Fortran 77 935, 946
compilers 2/viii, 2/xiv, 1364
compiling 936
conformable arrays 942f., 1094
CONTAINS statement 954, 957, 985, 1067, 1134, 1202
control structure 2/xiv, 959, 1219, 1305
conversion elemental functions 946
count() intrinsic function 948
cshift() intrinsic function 950, 969
cycle statement 959, 1219
data hiding 956ff., 1209
data parallelism 964
DATA statement 959
data types 937, 1336, 1346, 1361
deallocate statement 938f., 953f., 1197, 1266, 1293
deallocating array 938, 953f., 1197, 1266, 1293
defined types 956
deprecated features 947, 1057, 1161, 1256, 1286
derived types 937, 955
dimensional expansion 965ff.
do-loop 2/xiv
dot_product() intrinsic function 945, 949, 969, 1216
dynamical allocation of storage 2/xiii, 938, 941f., 953ff., 1327, 1336
elemental functions 940, 942, 946f., 951, 1015, 1083, 1364
elsewhere construct 943
eoshift() intrinsic function 950, 969, 1019f., 1078
epsilon() intrinsic function 951, 1189
evolution 2/xivff., 959, 987f.
example 936
exit statement 959, 1219
exponent() intrinsic function 1107
floor() intrinsic function 948
Fortran tip icon 1009
garbage collection 956
gather-scatter operations 2/xiif., 969, 981, 984, 1002, 1032, 1034, 1250
generic interface 2/xiii, 1083
generic procedures 939, 1015, 1083, 1094, 1096, 1364
global variables 955, 957, 1210
history 2/xff.
huge() intrinsic function 951
iand() intrinsic function 951
ibclr() intrinsic function 951
ibits() intrinsic function 951
ibset() intrinsic function 951
ieor() intrinsic function 951
IMPLICIT NONE statement 2/xiv, 936
implied do-list 968, 971, 1127
index loss 967f.
initialization expression 943, 959, 1012, 1127
inquiry functions 948
integer model 1144, 1149, 1156
INTENT attribute 1072, 1092
interface 939, 942, 1067, 1084, 1384
internal subprogram 2/xii, 2/xiv, 957, 1057, 1067, 1202f., 1256, 1302
interprocessor communication 969, 981, 1250
intrinsic data types 937

intrinsic procedures 939, 945ff., 987, 1016
ior() intrinsic function 951
ishft() intrinsic function 951
ishftc() intrinsic function 951
ISO (International Standards Organization) 2/xf., 2/xiiif.
keyword argument 2/xiv, 947f., 1341
kind() intrinsic function 951
KIND parameter 937, 946, 1125, 1144, 1192, 1254, 1261, 1284, 1361
language features 935ff.
lbound() intrinsic function 949
lexical comparison 952
linear algebra 969f., 1000ff., 1018f., 1026, 1040, 1200, 1326
linear recurrence 971, 988
linking 936
literal constant 937, 1361
logo for tips 2/viii, 1009
mask 948, 967f., 1006f., 1038, 1102, 1200, 1226, 1305, 1333f., 1368, 1378, 1382
matmul() intrinsic function 945, 949, 969, 1026, 1040, 1050, 1076, 1200, 1216, 1290, 1326
maxexponent() intrinsic function 1107
maxloc() intrinsic function 949, 961, 992f., 1015
maxval() intrinsic function 945, 948, 961, 1016, 1273
memory leaks 953, 956, 1327
memory management 938, 953ff.
merge() intrinsic function 945, 950, 1010, 1094f., 1099f.
Metcalf and Reid (M&R) 935
minloc() intrinsic function 949, 961, 992f.
minval() intrinsic function 948, 961
missing language features 983ff., 987ff.
modularization 956f.
MODULE facility 2/xiii, 936f., 939f., 953f., 957, 1067, 1298, 1320, 1322, 1324, 1330, 1346
MODULE subprograms 940
modulo() intrinsic function 946, 1156
named constant 940, 1012, 1361
named control structure 959, 1219, 1305
nearest() intrinsic function 952, 1146
nested where construct forbidden 943
not() intrinsic function 951
nullify statement 953f., 1070, 1302
numerical representation functions 951
ONLY option 941, 957, 1067
operator overloading 2/xiif.
operator, user-defined 2/xii
optional argument 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
outer product 969f.
overloading 940, 1083, 1102
pack() intrinsic function 945, 950, 964, 969, 991, 1170, 1176, 1178
pack, for selective evaluation 1087
parallel extensions 2/xv, 959ff., 964, 981, 984, 987, 1002, 1032
parallel programming 963ff.
PARAMETER attribute 1012
pointer 2/xiiif., 938f., 941, 944f., 952ff., 1067, 1070, 1197, 1210, 1212, 1266, 1302, 1327, 1336
pointer to function (missing) 1067
portability 963
present() intrinsic function 952
PRIVATE attribute 957, 1067
product() intrinsic function 948
programming conventions 937
PUBLIC attribute 957, 1067
quick start 936
radix() intrinsic function 1231
random_number() intrinsic function 1141, 1143
random_seed() intrinsic function 1141
real() intrinsic function 947, 1125
RECURSIVE keyword 958, 1065, 1067
recursive procedure 2/xiv, 958, 1065, 1067, 1166
reduction functions 948
reshape() intrinsic function 950, 969, 1247
RESULT keyword 958, 1073
SAVE attribute 953f., 958f., 1052, 1070, 1266, 1293
scale() intrinsic function 1107
scatter-with-combine (missing function) 984
scope 956ff.
scoping units 939
select case statement 2/xiv, 1010, 1036
shape() intrinsic function 938, 949
size() intrinsic function 938, 942, 945, 948
skew sections 985
sparse matrix representation 1030
specification statement 2/xiv
spread() intrinsic function 945, 950, 966ff., 969, 1000, 1094, 1290f.
statement functions deprecated 1057
stride (of an array) 944
structure constructor 2/xii
subscript triplet 944
sum() intrinsic function 945, 948, 966
tiny() intrinsic function 952
transformational functions 948
transpose() intrinsic function 950, 960, 969, 981, 1247
tricks 1009, 1072, 1146, 1274, 1278, 1280
truncation elemental functions 946
type checking 1140
ubound() intrinsic function 949
undefined pointer 953
unpack() intrinsic function 950, 964, 969
USE statement 936, 939f., 954, 957, 1067, 1384
utility functions 987ff.
vector subscripts 2/xiif., 969, 981, 984, 1002, 1032, 1034, 1250
visibility 956ff., 1209, 1293, 1296
WG5 technical committee 2/xi, 2/xiii, 2/xvf.
where construct 943, 985, 1060, 1291
X3J3 Committee 2/viii, 2/xff., 2/xv, 947, 959, 964, 968, 990
zero-length array 944

see also Intrinsic procedures
see also Fortran
Fortran 95 947, 959ff.
allocatable variables 961
blocks 960
cpu_time() intrinsic function 961
elemental functions 2/xiii, 2/xv, 940, 961, 986, 1015, 1083f., 1097f.
forall statement 2/xii, 2/xv, 960, 964, 968, 986, 1007
initialization of derived data type 2/xv
initialization of pointer 2/xv, 961
minor changes from Fortran 90 961
modified intrinsic functions 961
nested where construct 2/xv, 960, 1100
pointer association status 961
pointers 961
PURE attribute 2/xv, 960f., 964, 986
SAVE attribute 961
side effects 960
and skew array section 945, 985
see also Fortran
Fortran 2000 2/xvi
Forward deflation 363
Forward difference operator 161
Forward Euler differencing 826f.
Forward Time Centered Space *see* FTCS
Four-step framework, for FFT 983, 1239
Fourier analysis and cyclic reduction (FACR) 848f., 854
Fourier integrals
attenuation factors 583, 1261
endpoint corrections 578f., 1261
tail integration by parts 583
use of fast Fourier transform (FFT) 577ff., 1261ff.
Fourier transform 99, 490ff., 1235ff.
aliasing 495, 569
approximation of Dawson's integral 253
autocorrelation 492
basis functions compared 508f.
contrasted with wavelet transform 584, 594
convolution 492, 503f., 531ff., 909, 1253, 1354
correlation 492, 538f., 1254
cosine transform 190, 511ff., 851, 1245f.
cosine transform, second form 513, 852, 1246
critical sampling 494, 543, 545
definition 490
discrete Fourier transform (DFT) 184, 495ff.
Gaussian function 600
image processing 803, 805
infinite range 583
inverse of discrete Fourier transform 497
method for partial differential equations 848ff.
missing data 569
missing data, fast algorithm 574f., 1259
Nyquist frequency 494ff., 520, 543, 545, 569, 571
optimal (Wiener) filtering 539ff., 558
Parseval's theorem 492, 498, 544
power spectral density (PSD) 492f.
power spectrum estimation by FFT 542ff., 1254ff.
power spectrum estimation by maximum entropy method 565ff., 1258
properties of 491f.
sampling theorem 495, 543, 545, 600
scalings of 491
significance of a peak in 570
sine transform 508ff., 850, 1245
symmetries of 491
uneven sampling, fast algorithm 574f., 1259
unevenly sampled data 569ff., 574, 1258
and wavelets 592f.
Wiener-Khinchin theorem 492, 558, 566f.
see also Fast Fourier transform (FFT); Spectral density
Fractal region 360f.
Fractional step methods 847f.
Fredholm alternative 780
Fredholm equations 779f.
eigenvalue problems 780, 785
error estimate in solution 784
first kind 779
Fredholm alternative 780
homogeneous, second kind 785, 1325
homogeneous vs. inhomogeneous 779f.
ill-conditioned 780
infinite range 789
inverse problems 780, 795ff.
kernel 779f.
nonlinear 781
Nystrom method 782ff., 789, 1325
product Nystrom method 789, 1328ff.
second kind 779f., 782ff., 1325, 1331
with singularities 788, 1328ff.
with singularities, worked example 792, 1328ff.
subtraction of singularity 789
symmetric kernel 785
see also Inverse problems
Frequency domain 490
Frequency spectrum *see* Fast Fourier transform (FFT)
Frequentist, contrasted with Bayesian 810
Fresnel integrals 248ff.
asymptotic form 249
continued fraction 248f.
routine for 249f., 1123
series 248
Friday the Thirteenth 14f., 1011f.
FTCS (forward time centered space) 827ff., 839ff., 843
stability of 827ff., 839ff., 855
Full approximation storage (FAS) algorithm 874, 1339ff.
Full moon 14f., 936, 1011f.
Full multigrid method (FMG) 863, 868, 1334ff.
Full Newton methods, nonlinear least squares 683
Full pivoting 29, 1014
Full weighting 867
Function
Airy 204, 243f., 1121

approximation 99ff., 184ff., 1043, 1076ff.
associated Legendre polynomial 246ff., 764, 1122f., 1319
autocorrelation of 492
bandwidth limited 495
Bessel 172, 204, 223ff., 234, 1101ff., 1115ff.
beta 209, 1089
binomial coefficients 208f., 1087f.
branch cuts of 202f.
chi-square probability 215, 798
complex 202
confluent hypergeometric 204, 239
convolution of 492
correlation of 492
cosine integral 250f., 1123f.
Coulomb wave 204, 234
cumulative binomial probability 222f.
cumulative Poisson 209ff.
Dawson's integral 252ff., 600, 1127f.
digamma 216
elliptic integrals 254ff., 906, 1128ff.
error 213f., 248, 252, 601, 631, 635, 1094f., 1127, 1276f.
evaluation 159ff., 1070ff.
evaluation by path integration 201ff., 263, 1138
exponential integral 172, 215ff., 250, 1096f.
F-distribution probability 222
Fresnel integral 248ff., 1123
gamma 206, 1085
hypergeometric 202f., 263ff., 1138ff.
incomplete beta 219ff., 610, 1098ff., 1269
incomplete gamma 209ff., 615, 654, 657f., 1089ff., 1272, 1285
inverse hyperbolic 178, 255
inverse trigonometric 255
Jacobian elliptic 261, 1137f.
Kolmogorov-Smirnov probability 618f., 640, 1274, 1281
Legendre polynomial 172, 246, 674, 1122, 1291
logarithm 255
modified Bessel 229ff., 1109ff.
modified Bessel, fractional order 239ff., 1118ff.
overloading 1083
parallel evaluation 986, 1009, 1084, 1087, 1090, 1102, 1128, 1134
path integration to evaluate 201ff.
pathological 99f., 343
Poisson cumulant 214
representations of 490
routine for plotting a 342, 1182
sine and cosine integrals 248, 250ff., 1125f.
sn, dn, cn 261, 1137f.
spherical harmonics 246ff., 1122
spheroidal harmonic 764ff., 770ff., 1319ff., 1323ff.
Student's probability 221f.
variable number of arguments 1022
Weber 204
Functional iteration, for implicit equations 740f.
FWHM (full width at half maximum) 548f.

Gamma deviate 282f., 1153f.
Gamma function 206ff., 1085
  incomplete *see* Incomplete gamma function
Garbage collection 956
Gather-scatter operations 2/xiif., 984, 1002, 1032, 1034
  communication bottleneck 969, 981, 1250
  many-to-one 984, 1002, 1032, 1034
Gauss-Chebyshev integration 141, 144, 512f.
Gauss-Hermite integration 144, 789
  abscissas and weights 147, 1062
  normalization 147
Gauss-Jacobi integration 144
  abscissas and weights 148, 1063
Gauss-Jordan elimination 27ff., 33, 64, 1014f.
  operation count 34, 39
  solution of normal equations 667, 1288
  storage requirements 30
Gauss-Kronrod quadrature 154
Gauss-Laguerre integration 144, 789, 1060
Gauss-Legendre integration 145f., 1059
  see also Gaussian integration
Gauss-Lobatto quadrature 154, 190, 512
Gauss-Radau quadrature 154
Gauss-Seidel method (relaxation) 855, 857, 864ff., 1338
  nonlinear 876, 1341
Gauss transformation 256
Gaussian (normal) distribution 267, 652, 798
  central limit theorem 652f.
  deviates from 279f., 571, 1152
  kurtosis of 606
  multivariate 690
  semi-invariants of 608
  tails compared to Poisson 653
  two-dimensional (binormal) 631
  variance of skewness of 606
Gaussian elimination 33f., 51, 55, 1014f.
  fill-in 45, 64
  integral equations 786, 1326
  operation count 34
  outer product variant 1017
  in reduction to Hessenberg form 478, 1231
  relaxation solution of boundary value problems 753ff., 777, 1316
Gaussian function
  Hardy's theorem on Fourier transforms 600
  see also Gaussian (normal) distribution
Gaussian integration 127, 140ff., 789, 1059ff.
  calculation of abscissas and weights 142ff., 1009, 1059ff.
  error estimate in solution 784
  extensions of 153f.
  Golub-Welsch algorithm for weights and abscissas 150, 1064
  for integral equations 781, 783, 1325
  from known recurrence relation 150, 1064

nonclassical weight function 151ff., 788f., 1064f., 1328f.
and orthogonal polynomials 142, 1009, 1061
parallel calculation of formulas 1009, 1061
preassigned nodes 153f.
weight function $\log x$ 153
weight functions 140ff., 788f., 1059ff., 1328f.
Gear's method (stiff ODEs) 730
Geiger counter 266
Generalized eigenvalue problems 455
Generalized minimum residual method (GMRES) 78
Generic interface *see* Interface, generic
Generic procedures 939, 1083, 1094, 1096, 1364
elemental 940, 942, 946f., 1015, 1083
Geometric progression 972, 996f., 1365, 1372ff.
geop() utility function 972, 974, 989, 996, 1127
Geophysics, use of Backus-Gilbert method 809
Gerchberg-Saxton algorithm 805
get_diag() utility function 985, 989, 1005, 1226
Gilbert and Sullivan 714
Givens reduction 462f., 473
fast 463
operation count 463
Glassman, A.J. 180
Global optimization 387f., 436ff., 650, 1219ff.
continuous variables 443f., 1222
Global variables 940, 953f., 1210
allocatable array method 954, 1197, 1212, 1266, 1287, 1298
communicated via internal subprogram 954, 957f., 1067, 1226
danger of 957, 1209, 1293, 1296
pointer method 954, 1197, 1212, 1266, 1287, 1302
Globally convergent
minimization 418ff., 1215
root finding 373, 376ff., 382, 749f., 752, 1196, 1314f.
GMRES (generalized minimum residual method) 78
GNU Emacs 1/xvi
Godunov's method 837
Golden mean (golden ratio) 21, 349, 392f., 399
Golden section search 341, 389ff., 395, 1202ff.
Golub-Welsch algorithm, for Gaussian quadrature 150, 1064
Goodness-of-fit 650, 654, 657f., 662, 690, 1285
GOTO statements, danger of 9, 959
Gram-Schmidt
biorthogonalization 415f.
orthogonalization 94, 450f., 1039
SVD as alternative to 58
Graphics, function plotting 342, 1182f.
Gravitational potential 519
Gray code 300, 881, 886ff., 1344
Greenbaum, A. 79
Gregorian calendar 13, 16, 1011, 1013
Grid square 116f.
Group, dihedral 894, 1345
Guard digits 882, 1343

Half weighting 867, 1337
Halton's quasi-random sequence 300
Hamming window 547
Hamming's motto 341
Hann window 547
Harmonic analysis *see* Fourier transform
Hashing 293, 1144, 1148, 1156
for random number seeds 1147f.
HDLC checksum 890
Heap (data structure) 327f., 336, 897, 1179
Heapsort 320, 327f., 336, 1171f., 1179
Helmholtz equation 852
Hermite polynomials 144, 147
approximation of roots 1062
Hermitian matrix 450ff., 475
Hertz (unit of frequency) 490
Hessenberg matrix 94, 453, 470, 476ff., 488, 1231
see also Matrix
Hessian matrix 382, 408, 415f., 419f., 676ff., 803, 815
is inverse of covariance matrix 667, 679
second derivatives in 676
Hexadecimal constants 17f., 276, 293
initialization 959
Hierarchically band diagonal matrix 598
Hierarchy of program structure 6ff.
High-order not same as high-accuracy 100f., 124, 389, 399, 705, 709, 741
High-pass filter 551
High-Performance Fortran (HPF) 2/xvf., 964, 981, 984
scatter-with-add 1032
Hilbert matrix 83
Home page, Numerical Recipes 1/xx, 2/xvii
Homogeneous linear equations 53
Hook step methods 386
Hotelling's method for matrix inverse 49, 598
Householder transformation 52, 453, 462ff., 469, 473, 475, 478, 481ff., 1227f.
operation count 467
in QR decomposition 92, 1039
HPF *see* High-Performance Fortran
Huffman coding 564, 881, 896f., 902, 1346ff.
huge() intrinsic function 951
Hyperbolic functions, explicit formulas for inverse 178
Hyperbolic partial differential equations 818
advective equation 826
flux-conservative initial value problems 825ff.
Hypergeometric function 202f., 263ff.
routine for 264f., 1138
Hypothesis, null 603

I2B, defined 937

I4B, defined 937
iand() intrinsic function 951
ibclr() intrinsic function 951
ibits() intrinsic function 951
IBM 1/xxiii, 2/xix
  bad random number generator 268
  Fortran 90 compiler 2/viii
  PC 4, 276, 293, 886
  PC-RT 4
  radix base for floating point arithmetic 476
  RS6000 2/viii, 4
IBM checksum 894
ibset() intrinsic function 951
ICCG (incomplete Cholesky conjugate gradient method) 824
ICF (intrinsic correlation function) model 817
Identity (unit) matrix 25
IEEE floating point format 276, 882f., 1343
ieor() intrinsic function 951
if statement, arithmetic 2/xi
if structure 12f.
ifirstloc() utility function 989, 993, 1041, 1346
IIR (infinite impulse response) filter 552ff., 566
Ill-conditioned integral equations 780
Image processing 519, 803
  cosine transform 513
  fast Fourier transform (FFT) 519, 523, 803
  as an inverse problem 803
  maximum entropy method (MEM) 809ff.
  from modulus of Fourier transform 805
  wavelet transform 596f., 1267f.
imaxloc() utility function 989, 993, 1017
iminloc() utility function 989, 993, 1046, 1076
Implicit
  function theorem 340
  pivoting 30, 1014
  shifts in QL method 472ff.
Implicit differencing 827
  for diffusion equation 840
  for stiff equations 729, 740, 1308
IMPLICIT NONE statement 2/xiv, 936
Implied do-list 968, 971, 1127
Importance sampling, in Monte Carlo 306f.
Improper integrals 135ff., 1055
Impulse response function 531, 540, 552
IMSL 1/xxiii, 2/xx, 26, 64, 205, 364, 369, 454
In-place selection 335, 1178f.
Included file, superseded by module 940
Incomplete beta function 219ff., 1098ff.
  for F-test 613, 1271
  routine for 220f., 1097
  for Student's t 610, 613, 1269
Incomplete Cholesky conjugate gradient method (ICCG) 824
Incomplete gamma function 209ff., 1089ff.
  for chi-square 615, 654, 657f., 1272, 1285
  deviates from 282f., 1153
  in mode estimation 610
  routine for 211f., 1089
Increment of linear congruential generator 268
Indentation of blocks 9
Index 934ff., 1446ff.
  this entry 1464
Index loss 967f., 1038
Index table 320, 329f., 1173ff., 1176
Inequality constraints 423
Inheritance 8
Initial value problems 702, 818f.
  see also Differential equations; Partial differential equations
Initialization of derived data type 2/xv
Initialization expression 943, 959, 1012, 1127
Injection operator 864, 1337
Instability *see* Stability
Integer model, in Fortran 90 1144, 1149, 1156
Integer programming 436
Integral equations 779ff.
  adaptive stepsize control 788
  block-by-block method 788
  correspondence with linear algebraic equations 779ff.
  degenerate kernel 785
  eigenvalue problems 780, 785
  error estimate in solution 784
  Fredholm 779f., 782ff., 1325, 1331
  Fredholm alternative 780
  homogeneous, second kind 785, 1325
  ill-conditioned 780
  infinite range 789
  inverse problems 780, 795ff.
  kernel 779
  nonlinear 781, 787
  Nystrom method 782ff., 789, 1325
  product Nystrom method 789, 1328ff.
  with singularities 788ff., 1328ff.
  with singularities, worked example 792, 1328ff.
  subtraction of singularity 789
  symmetric kernel 785
  unstable quadrature 787f.
  Volterra 780f., 786ff., 1326f.
  wavelets 782
  see also Inverse problems
Integral operator, wavelet approximation of 597, 782
Integration of functions 123ff., 1052ff.
  cosine integrals 250, 1125
  Fourier integrals 577ff., 1261
  Fourier integrals, infinite range 583
  Fresnel integrals 248, 1123
  Gauss-Hermite 147f., 1062
  Gauss-Jacobi 148, 1063
  Gauss-Laguerre 146, 1060
  Gauss-Legendre 145, 1059
  integrals that are elliptic integrals 254
  path integration 201ff.
  sine integrals 250, 1125
  see also Quadrature
Integro-differential equations 782
INTENT attribute 1072, 1092
Interface (Fortran 90) 939, 942, 1067

for communication between program parts 957, 1209, 1293, 1296
explicit 939, 942, 1067, 1384
generic 2/xiii, 940, 1015, 1083, 1094, 1096
implicit 939
for Numerical Recipes 1384ff.
Interface block 939, 1084, 1384
Interface, in programs 2, 8
Intermediate value theorem 343
Internal subprogram (Fortran 90) 2/xiv, 954, 957, 1067, 1202f., 1226
nesting of 2/xii
resembles C macro 1302
supersedes statement function 1057, 1256
International Standards Organization (ISO) 2/xf., 2/xiii
Internet, availability of code over 1/xx, 2/xvii
Interpolation 99ff.
Aitken's algorithm 102
avoid 2-stage method 100
avoid in Fourier analysis 569
bicubic 118f., 1049f.
bilinear 117
caution on high-order 100
coefficients of polynomial 100, 113ff., 191, 575, 1047f., 1078
for computing Fourier integrals 578
error estimates for 100
of functions with poles 104ff., 1043f.
inverse quadratic 353, 395ff., 1204
multidimensional 101f., 116ff., 1049ff.
in multigrid method 866, 1337
Neville's algorithm 102f., 182, 1043
Nystrom 783, 1326
offset arrays 104, 113
operation count for 100
operator 864, 1337
order of 100
and ordinary differential equations 101
oscillations of polynomial 100, 116, 389, 399
parabolic, for minimum finding 395, 1204
polynomial 99, 102ff., 182, 1043
rational Chebyshev approximation 197ff., 1081
rational function 99, 104ff., 194ff., 225, 718ff., 726, 1043f., 1080, 1306
reverse (extirpolation) 574, 1261
spline 100, 107ff., 120f., 1044f., 1050f.
trigonometric 99
see also Fitting
Interprocessor communication 969, 981
Interval variable (statistics) 623
Intrinsic correlation function (ICF) model 817
Intrinsic data types 937
Intrinsic procedures
array inquiry 938, 942, 948ff.
array manipulation 950
array reduction 948
array unary and binary functions 949
backwards-compatibility 946
bit manipulation 2/xiii, 951
character 952
cmplx 1254
conversion elemental 946
elemental 940, 942, 946f., 951, 1083, 1364
generic 939, 1083f., 1364
lexical comparison 952
numeric inquiry 2/xiv, 1107, 1231, 1343
numerical 946, 951f.
numerical representation 951
pack used for sorting 1171
random_number 1143
real 1254
top 10 945
truncation 946f.
see also Fortran 90
Inverse hyperbolic function 178, 255
Inverse iteration *see* Eigensystems
Inverse problems 779, 795ff.
Backus-Gilbert method 806ff.
Bayesian approach 799, 810f., 816f.
central idea 799
constrained linear inversion method 799ff.
data inversion 807
deterministic constraints 804ff.
in geophysics 809
Gerchberg-Saxton algorithm 805
incomplete Fourier coefficients 813
and integral equations 780
linear regularization 799ff.
maximum entropy method (MEM) 810, 815f.
MEM demystified 814
Phillips-Twomey method 799ff.
principal solution 797
regularization 796ff.
regularizing operator 798
stabilizing functional 798
Tikhonov-Miller regularization 799ff.
trade-off curve 795
trade-off curve, Backus-Gilbert method 809
two-dimensional regularization 803
use of conjugate gradient minimization 804, 815
use of convex sets 804
use of Fourier transform 803, 805
Van Cittert's method 804
Inverse quadratic interpolation 353, 395ff., 1204
Inverse response kernel, in Backus-Gilbert method 807
Inverse trigonometric function 255
ior() intrinsic function 951
ISBN (International Standard Book Number) checksum 894
ishft() intrinsic function 951
ishftc() intrinsic function 951
ISO (International Standards Organization) 2/xf., 2/xiii
Iterated integrals 155
Iteration 9f.
functional 740f.
to improve solution of linear algebraic equations 47ff., 195, 1022
for linear algebraic equations 26

required for two-point boundary value problems 745
in root finding 340f.
Iteration matrix 856
ITPACK 71
Iverson, John 2/xi

Jacobi matrix, for Gaussian quadrature 150, 1064
Jacobi polynomials, approximation of roots 1064
Jacobi transformation (or rotation) 94, 453, 456ff., 462, 475, 489, 1041, 1225
Jacobian determinant 279, 774
Jacobian elliptic functions 261, 1137f.
Jacobian matrix 374, 376, 379, 382, 731, 1197f., 1309
singular in Newton's rule 386
Jacobi's method (relaxation) 855ff., 864
Jenkins-Traub method 369
Julian Day 1, 13, 16, 936, 1010ff.
Jump transposition errors 895

K-S test *see* Kolmogorov-Smirnov test
Kalman filter 700
Kanji 2/xii
Kaps-Rentrop method 730, 1308
Kendall's tau 634, 637ff., 1279
Kennedy, Ken 2/xv
Kepler's equation 1061
Kermit checksum 889
Kernel 779
averaging, in Backus-Gilbert method 807
degenerate 785
finite rank 785
inverse response 807
separable 785
singular 788f., 1328
symmetric 785
Keys used in sorting 329, 889
Keyword argument 2/xiv, 947f., 1341
kind() intrinsic function 951
KIND parameter 946, 1261, 1284
and cmplx() intrinsic function 1125, 1192, 1254
default 937
for Numerical Recipes 1361
for random numbers 1144
and real() intrinsic function 1125
Kolmogorov-Smirnov test 614, 617ff., 694, 1273f.
two-dimensional 640, 1281ff.
variants 620ff., 640, 1281
Kuiper's statistic 621
Kurtosis 606, 608, 1269

L-estimate 694
Labels, statement 9
Lag 492, 538, 553
Lagged Fibonacci generator 1142, 1148ff.
Lagrange multiplier 795
Lagrange's formula for polynomial interpolation 84, 102f., 575, 578
Laguerre polynomials, approximation of roots 1061
Laguerre's method 341, 365f., 1191f.
Lanczos lemma 498f.
Lanczos method for gamma function 206, 1085
Landen transformation 256
LAPACK 26, 1230
Laplace's equation 246, 818
see also Poisson equation
Las Vegas 625
Latin square or hypercube 305f.
Laurent series 566
Lax method 828ff., 836, 845f.
multidimensional 845f.
Lax-Wendroff method 835ff.
lbound() intrinsic function 949
Leakage in power spectrum estimation 544, 548
Leakage width 548f.
Leapfrog method 833f.
Least squares filters *see* Savitzky-Golay filters
Least squares fitting 645, 651ff., 655ff., 660ff., 665ff., 1285f., 1288f.
contrasted to general minimization problems 684ff.
degeneracies in 671f., 674
Fourier components 570
as M-estimate for normal errors 696
as maximum likelihood estimator 652
as method for smoothing data 645, 1283
Fourier components 1258
freezing parameters in 668, 700
general linear case 665ff., 1288, 1290f.
Levenberg-Marquardt method 678ff., 816, 1292f.
Lomb periodogram 570, 1258
multidimensional 675
nonlinear 386, 675ff., 816, 1292
nonlinear, advanced methods 683
normal equations 645, 666f., 800, 1288
normal equations often singular 670, 674
optimal (Wiener) filtering 540f.
QR method in 94, 668
for rational Chebyshev approximation 199f., 1081f.
relation to linear correlation 630, 658
Savitzky-Golay filter as 645, 1283
singular value decomposition (SVD) 25f., 51ff., 199f., 670ff., 1081, 1290
skewed by outliers 653
for spectral analysis 570, 1258
standard (probable) errors on fitted parameters 667, 671
weighted 652
see also Fitting
L'Ecuyer's long period random generator 271, 273
Least squares fitting
standard (probable) errors on fitted parameters 1288, 1290
weighted 1285
Left eigenvalues or eigenvectors 451
Legal matters 1/xx, 2/xvii
Legendre elliptic integral *see* Elliptic integrals

Legendre polynomials 246, 1122
fitting data to 674, 1291f.
recurrence relation 172
shifted monic 151
see also Associated Legendre polynomials; Spherical harmonics
Lehmer-Schur algorithm 369
Lemarie's wavelet 593
Lentz's method for continued fraction 165, 212
Lepage, P. 309
Leptokurtic distribution 606
Levenberg-Marquardt algorithm 386, 678ff., 816, 1292
advanced implementation 683
Levinson's method 86, 1038
Lewis, H.W. 275
Lexical comparison functions 952
LGT, defined 937
License information 1/xx, 2/xviiff.
Limbo 356
Limit cycle, in Laguerre's method 365
Line minimization *see* Minimization, along a ray
Line search *see* Minimization, along a ray
Linear algebra, intrinsic functions for parallelization 969f., 1026, 1040, 1200, 1326
Linear algebraic equations 22ff., 1014
band diagonal 43ff., 1019
biconjugate gradient method 77, 1034ff.
Cholesky decomposition 89f., 423, 455, 668, 1038f.
complex 41
computing $\mathbf{A}^{-1} \cdot \mathbf{B}$ 40
conjugate gradient method 77ff., 599, 1034
cyclic tridiagonal 67, 1030
direct methods 26, 64, 1014, 1030
Fortran 90 vs. library routines 1016
Gauss-Jordan elimination 27ff., 1014
Gaussian elimination 33f., 1014f.
Hilbert matrix 83
Hotelling's method 49, 598
and integral equations 779ff., 783, 1325
iterative improvement 47ff., 195, 1022
iterative methods 26, 77ff., 1034
large sets of 23
least squares solution 53ff., 57f., 199f., 671, 1081, 1290
LU decomposition 34ff., 195, 386, 732, 783, 786, 801, 1016, 1022, 1325f.
nonsingular 23
overdetermined 25f., 199, 670, 797
partitioned 70
QR decomposition 91f., 382, 386, 668, 1039f., 1199
row vs. column elimination 31f.
Schultz's method 49, 598
Sherman-Morrison formula 65ff., 83
singular 22, 53, 58, 199, 670
singular value decomposition (SVD) 51ff., 199f., 670ff., 797, 1022, 1081, 1290
sparse 23, 43, 63ff., 732, 804, 1020f., 1030
summary of tasks 25f.
Toeplitz 82, 85ff., 195, 1038
tridiagonal 26, 42f., 64, 109, 150, 453f., 462ff., 469ff., 488, 839f., 853, 861f., 1018f., 1227ff.
Vandermonde 82ff., 114, 1037, 1047
wavelet solution 597ff., 782
Woodbury formula 68ff., 83
see also Eigensystems
Linear congruential random number generator 267ff., 1142
choice of constants for 274ff.
Linear constraints 423
Linear convergence 346, 393
Linear correlation (statistics) 630ff., 1276
Linear dependency
constructing orthonormal basis 58, 94
of directions in N-dimensional space 409
in linear algebraic equations 22f.
Linear equations *see* Differential equations; Integral equations; Linear algebraic equations
Linear inversion method, constrained 799ff.
Linear prediction 557ff.
characteristic polynomial 559
coefficients 557ff., 1256
compared to maximum entropy method 558
compared with regularization 801
contrasted to polynomial extrapolation 560
related to optimal filtering 558
removal of bias in 563
stability 559f., 1257
Linear predictive coding (LPC) 563ff.
Linear programming 387, 423ff., 1216ff.
artificial variables 429
auxiliary objective function 430
basic variables 426
composite simplex algorithm 435
constraints 423
convergence criteria 432
degenerate feasible vector 429
dual problem 435
equality constraints 423
feasible basis vector 426
feasible vector 424
fundamental theorem 426
inequality constraints 423
left-hand variables 426
nonbasic variables 426
normal form 426
objective function 424
optimal feasible vector 424
pivot element 428f.
primal-dual algorithm 435
primal problem 435
reduction to normal form 429ff.
restricted normal form 426ff.
revised simplex method 435
right-hand variables 426
simplex method 402, 423ff., 431ff., 1216ff.
slack variables 429
tableau 427
vertex of simplex 426

Linear recurrence *see* Recurrence relation
Linear regression 655ff., 660ff., 1285ff.
see also Fitting
Linear regularization 799ff.
LINPACK 26
Literal constant 937, 1361
Little-endian 293
Local extrapolation 709
Local extremum 387f., 437
Localization of roots *see* Bracketing
Logarithmic function 255
Lomb periodogram method of spectral analysis 569f., 1258f.
fast algorithm 574f., 1259
Loops 9f.
Lorentzian probability distribution 282, 696f.
Low-pass filter 551, 644f., 1283f.
Lower subscript 944
lower_triangle() utility function 989, 1007, 1200
LP coefficients *see* Linear prediction
LPC (linear predictive coding) 563ff.
LU decomposition 34ff., 47f., 51, 55, 64, 97, 374, 667, 732, 1016, 1022
for $\mathbf{A}^{-1} \cdot \mathbf{B}$ 40
backsubstitution 39, 1017
band diagonal matrix 43ff., 1020
complex equations 41f.
Crout's algorithm 36ff., 45, 1017
for integral equations 783, 786, 1325f.
for inverse iteration of eigenvectors 488
for inverse problems 801
for matrix determinant 41
for matrix inverse 40, 1016
for nonlinear sets of equations 374, 386, 1196
operation count 36, 39
outer product Gaussian elimination 1017
for Padé approximant 195, 1080
pivoting 37f., 1017
repeated backsubstitution 40, 46
solution of linear algebraic equations 40, 1017
solution of normal equations 667
for Toeplitz matrix 87
Lucifer 290

***M*&R (Metcalf and Reid) 935**
M-estimates 694ff.
how to compute 697f.
local 695ff.
see also Maximum likelihood estimate
Machine accuracy 19f., 881f., 1189, 1343
Macintosh, *see* Apple Macintosh
Maehly's procedure 364, 371
Magic
in MEM image restoration 814
in Padé approximation 195
Mantissa in floating point format 19, 882, 909, 1343
Marginals 624
Marquardt method (least squares fitting) 678ff., 816, 1292f.
Marsaglia shift register 1142, 1148ff.
Marsaglia, G. 1142, 1149
mask 1006f., 1102, 1200, 1226, 1305, 1333f., 1368, 1378, 1382
optional argument 948
optional argument, facilitates parallelism 967f., 1038
Mass, center of 295ff.
MasterCard checksum 894
Mathematical Center (Amsterdam) 353
Mathematical intrinsic functions 946, 951f.
matmul() intrinsic function 945, 949, 969, 1026, 1040, 1050, 1076, 1200, 1216, 1290, 1326
Matrix 23ff.
add vector to diagonal 1004, 1234, 1366, 1381
approximation of 58f., 598f.
band diagonal 42ff., 64, 1019
band triangular 64
banded 26, 454
bidiagonal 52
block diagonal 64, 754
block triangular 64
block tridiagonal 64
bordered 64
characteristic polynomial 449, 469
Cholesky decomposition 89f., 423, 455, 668, 1038f.
column augmented 28, 1014
complex 41
condition number 53, 78
create unit matrix 1006, 1382
curvature 677
cyclic banded 64
cyclic tridiagonal 67, 1030
defective 450, 476, 489
of derivatives *see* Hessian matrix; Jacobian determinant
design (fitting) 645, 665, 801, 1082
determinant of 25, 41
diagonal of sparse matrix 1033ff.
diagonalization 452ff., 1225ff.
elementary row and column operations 28f.
finite differencing of partial differential equations 821ff.
get diagonal 985, 1005, 1226f., 1366, 1381f.
Hermitian 450, 454, 475
Hermitian conjugate 450
Hessenberg 94, 453, 470, 476ff., 488, 1231ff.
Hessian *see* Hessian matrix
hierarchically band diagonal 598
Hilbert 83
identity 25
ill-conditioned 53, 56, 114
indexed storage of 71f., 1030
and integral equations 779, 783, 1325
inverse 25, 27, 34, 40, 65ff., 70, 95ff., 1014, 1016f.
inverse, approximate 49
inverse by Hotelling's method 49, 598
inverse by Schultz's method 49, 598
inverse multiplied by a matrix 40
iteration for inverse 49, 598

Jacobi transformation 453, 456ff., 462, 1225f.
Jacobian 731, 1309
logical dimension 24
lower triangular 34f., 89, 781, 1016
lower triangular mask 1007, 1200, 1382
multiplication denoted by dot 23
multiplication, intrinsic function 949, 969, 1026, 1040, 1050, 1200, 1326
norm 50
normal 450ff.
nullity 53
nullspace 25, 53f., 449, 795
orthogonal 91, 450, 463ff., 587
orthogonal transformation 452, 463ff., 469, 1227
orthonormal basis 58, 94
outer product denoted by cross 66, 420
partitioning for determinant 70
partitioning for inverse 70
pattern multiply of sparse 74
physical dimension 24
positive definite 26, 89f., 668, 1038
QR decomposition 91f., 382, 386, 668, 1039, 1199
range 53
rank 53
residual 49
row and column indices 23
row vs. column operations 31f.
self-adjoint 450
set diagonal elements 1005, 1200, 1366, 1382
similarity transform 452ff., 456, 476, 478, 482
singular 53f., 58, 449
singular value decomposition 26, 51ff., 797
sparse 23, 63ff., 71, 598, 732, 754, 804, 1030ff.
special forms 26
splitting in relaxation method 856f.
spread 808
square root of 423, 455
symmetric 26, 89, 450, 454, 462ff., 668, 785, 1038, 1225, 1227
threshold multiply of sparse 74, 1031
Toeplitz 82, 85ff., 195, 1038
transpose() intrinsic function 950
transpose of sparse 73f., 1033
triangular 453
tridiagonal 26, 42f., 64, 109, 150, 453f., 462ff., 469ff., 488, 839f., 853, 861f., 1018f., 1227ff.
tridiagonal with fringes 822
unitary 450
updating 94, 382, 386, 1041, 1199
upper triangular 34f., 91, 1016
upper triangular mask 1006, 1226, 1305, 1382
Vandermonde 82ff., 114, 1037, 1047
see also Eigensystems
Matrix equations *see* Linear algebraic equations
Matterhorn 606
maxexponent() intrinsic function 1107
Maximization *see* Minimization
Maximum entropy method (MEM) 565ff., 1258
algorithms for image restoration 815f.
Bayesian 816f.
Cornwell-Evans algorithm 816
demystified 814
historic vs. Bayesian 816f.
image restoration 809ff.
intrinsic correlation function (ICF) model 817
for inverse problems 809ff.
operation count 567
see also Linear prediction
Maximum likelihood estimate (M-estimates) 690, 694ff.
and Bayes' Theorem 811
chi-square test 690
defined 652
how to compute 697f.
mean absolute deviation 696, 698, 1294
relation to least squares 652
maxloc() intrinsic function 949, 992f., 1015
modified in Fortran 95 961
maxval() intrinsic function 945, 948, 961, 1016, 1273
Maxwell's equations 825f.
Mean(s)
of distribution 604f., 608f., 1269
statistical differences between two 609ff., 1269f.
Mean absolute deviation of distribution 605, 696, 1294
related to median 698
Measurement errors 650
Median 320
calculating 333
of distribution 605, 608f.
as L-estimate 694
role in robust straight line fitting 698
by selection 698, 1294
Median-of-three, in Quicksort 324
MEM *see* Maximum entropy method (MEM)
Memory leak 953, 956, 1071, 1327
Memory management 938, 941f., 953ff., 1327, 1336
merge construct 945, 950, 1099f.
for conditional scalar expression 1010, 1094f.
contrasted with where 1023
parallelization 1011
Merge-with-dummy-values idiom 1090
Merit function 650
in general linear least squares 665
for inverse problems 797
nonlinear models 675
for straight line fitting 656, 698
for straight line fitting, errors in both coordinates 660, 1286
Mesh-drift instability 834f.
Mesokurtic distribution 606
Metcalf, Michael 2/viii
see also M&R
Method of regularization 799ff.

Metropolis algorithm 437f., 1219
Microsoft 1/xxii, 2/xix
Microsoft Fortran PowerStation 2/viii
Midpoint method *see* Modified midpoint method; Semi-implicit midpoint rule
Mikado, or Town of Titipu 714
Miller's algorithm 175, 228, 1106
MIMD machines (Multiple Instruction Multiple Data) 964, 985, 1071, 1084
Minimal solution of recurrence relation 174
Minimax polynomial 186, 198, 1076
Minimax rational function 198
Minimization 387ff.
  along a ray 77, 376f., 389, 406ff., 412f., 415f., 418, 1195f., 1211, 1213
  annealing, method of simulated 387f., 436ff., 1219ff.
  bracketing of minimum 390ff., 402, 1201f.
  Brent's method 389, 395ff., 399, 660f., 1204ff., 1286
  Broyden-Fletcher-Goldfarb-Shanno algorithm 390, 418ff., 1215
  chi-square 653ff., 675ff., 1285, 1292
  choice of methods 388f.
  combinatorial 436f., 1219
  conjugate gradient method 390, 413ff., 804, 815, 1210, 1214
  convergence rate 393, 409
  Davidon-Fletcher-Powell algorithm 390, 418ff., 1215
  degenerate 795
  direction-set methods 389, 406ff., 1210ff.
  downhill simplex method 389, 402ff., 444, 697f., 1208, 1222ff.
  finding best-fit parameters 650
  Fletcher-Reeves algorithm 390, 414ff., 1214
  functional 795
  global 387f., 443f., 650, 1219, 1222
  globally convergent multidimensional 418, 1215
  golden section search 390ff., 395, 1202ff.
  multidimensional 388f., 402ff., 1208ff., 1214
  in nonlinear model fitting 675f., 1292
  Polak-Ribiere algorithm 389, 414ff., 1214
  Powell's method 389, 402, 406ff., 1210ff.
  quasi-Newton methods 376, 390, 418ff., 1215
  and root finding 375
  scaling of variables 420
  by searching smaller subspaces 815
  steepest descent method 414, 804
  termination criterion 392, 404
  use in finding double roots 341
  use for sparse linear systems 77ff.
  using derivatives 389f., 399ff., 1205ff.
  variable metric methods 390, 418ff., 1215
  see also Linear programming
Minimum residual method, for sparse system 78
minloc() intrinsic function 949, 992f.
  modified in Fortran 95 961
MINPACK 683
minval() intrinsic function 948, 961
MIPS 886
Missing data problem 569
Mississippi River 438f., 447
MMP (massively multiprocessor) machines 965ff., 974, 981, 984, 1016ff., 1021, 1045, 1226ff., 1250
Mode of distribution 605, 609
Modeling of data *see* Fitting
Model-trust region 386, 683
Modes, homogeneous, of recursive filters 554
Modified Bessel functions *see* Bessel functions
Modified Lentz's method, for continued fractions 165
Modified midpoint method 716ff., 720, 1302f.
Modified moments 152
Modula-2 7
Modular arithmetic, without overflow 269, 271, 275
Modular programming 2/xiii, 7f., 956ff., 1209, 1293, 1296, 1346
MODULE facility 2/xiii, 936f., 939f., 957, 1067, 1298, 1320, 1322, 1324, 1330, 1346
  initializing random number generator 1144ff.
  in nr.f90 936, 941f., 1362, 1384ff.
  in nrtype.f90 936f., 1361f.
  in nrutil.f90 936, 1070, 1362, 1364ff.
  sparse matrix 1031
  undefined variables on exit 953, 1266
Module subprogram 940
modulo() intrinsic function 946, 1156
Modulus of linear congruential generator 268
Moments
  of distribution 604ff., 1269
  filter that preserves 645
  modified problem of 151f.
  problem of 83
  and quadrature formulas 791, 1328
  semi-invariants 608
Monic polynomial 142f.
Monotonicity constraint, in upwind differencing 837
Monte Carlo 155ff., 267
  adaptive 306ff., 1161ff.
  bootstrap method 686f.
  comparison of sampling methods 309
  exploration of binary tree 290
  importance sampling 306f.
  integration 124, 155ff., 295ff., 306ff., 1161
  integration, recursive 314ff., 1164ff.
  integration, using Sobol' sequence 304
  integration, VEGAS algorithm 309ff., 1161
  and Kolmogorov-Smirnov statistic 622, 640
  partial differential equations 824
  quasi-random sequences in 299ff.
  quick and dirty 686f.
  recursive 306ff., 314ff., 1161, 1164ff.
  significance of Lomb periodogram 570
  simulation of data 654, 684ff., 690
  stratified sampling 308f., 314, 1164

Moon, calculate phases of 1f., 14f., 936, 1010f.
Mother functions 584
Mother Nature 684, 686
Moving average (MA) model 566
Moving window averaging 644
Mozart 9
MS 1/xxii, 2/xix
Muller's method 364, 372
Multidimensional
  confidence levels of fitting 688f.
  data, use of binning 623
  Fourier transform 515ff., 1241, 1246, 1251
  Fourier transform, real data 519ff., 1248f.
  initial value problems 844ff.
  integrals 124, 155ff., 295ff., 306ff., 1065ff., 1161ff.
  interpolation 116ff., 1049ff.
  Kolmogorov-Smirnov test 640, 1281
  least squares fitting 675
  minimization 402ff., 406ff., 413ff., 1208ff., 1214f., 1222ff.
  Monte Carlo integration 295ff., 306ff., 1161ff.
  normal (Gaussian) distribution 690
  optimization 388f.
  partial differential equations 844ff.
  root finding 340ff., 358, 370, 372ff., 746, 749f., 752, 754, 1194ff., 1314ff.
  search using quasi-random sequence 300
  secant method 373, 382f., 1199f.
  wavelet transform 595, 1267f.
Multigrid method 824, 862ff., 1334ff.
  avoid SOR 866
  boundary conditions 868f.
  choice of operators 868
  coarse-to-fine operator 864, 1337
  coarse-grid correction 864f.
  cycle 865
  dual viewpoint 875
  fine-to-coarse operator 864, 1337
  full approximation storage (FAS) algorithm 874, 1339ff.
  full multigrid method (FMG) 863, 868, 1334ff.
  full weighting 867
  Gauss-Seidel relaxation 865f., 1338
  half weighting 867, 1337
  importance of adjoint operator 867
  injection operator 864, 1337
  interpolation operator 864, 1337
  line relaxation 866
  local truncation error 875
  Newton's rule 874, 876, 1339, 1341
  nonlinear equations 874ff., 1339ff.
  nonlinear Gauss-Seidel relaxation 876, 1341
  odd-even ordering 866, 869, 1338
  operation count 862
  prolongation operator 864, 1337
  recursive nature 865, 1009, 1336
  relative truncation error 875
  relaxation as smoothing operator 865
  restriction operator 864, 1337
  speeding up FMG algorithm 873
  stopping criterion 875f.
  straight injection 867
  symbol of operator 866f.
  use of Richardson extrapolation 869
  V-cycle 865, 1336
  W-cycle 865, 1336
  zebra relaxation 866
Multiple precision arithmetic 906ff., 1352ff.
Multiple roots 341, 362
Multiplication, complex 171
Multiplication, multiple precision 907, 909, 1353f.
Multiplier of linear congruential generator 268
Multistep and multivalue methods (ODEs) 740ff.
  see also Differential Equations; Predictor-corrector methods
Multivariate normal distribution 690
Murphy's Law 407
Musical scores 5f.

NAG 1/xxiii, 2/xx, 26, 64, 205, 454
  Fortran 90 compiler 2/viii, 2/xiv
Named constant 940
  initialization 1012
  for Numerical Recipes 1361
Named control structure 959, 1219, 1305
National Science Foundation (U.S.) 1/xvii, 1/xix, 2/ix
Natural cubic spline 109, 1044f.
Navier-Stokes equation 830f.
nearest() intrinsic function 952, 1146
Needle, eye of (minimization) 403
Negation, multiple precision 907, 1353f.
Negentropy 811, 896
Nelder-Mead minimization method 389, 402, 1208
Nested iteration 868
Neumann boundary conditions 820, 840, 851, 858
Neutrino 640
Neville's algorithm 102f., 105, 134, 182, 1043
Newton-Cotes formulas 125ff., 140
Newton-Raphson method *see* Newton's rule
Newton's rule 143f., 180, 341, 355ff., 362, 364, 469, 1059, 1189
  with backtracking 376, 1196
  caution on use of numerical derivatives 356ff.
  fractal domain of convergence 360f.
  globally convergent multidimensional 373, 376ff., 382, 749f., 752, 1196, 1199, 1314f.
  for matrix inverse 49, 598
  in multidimensions 370, 372ff., 749f., 752, 754, 1194ff., 1314ff.
  in nonlinear multigrid 874, 876, 1339, 1341
  nonlinear Volterra equations 787
  for reciprocal of number 911, 1355
  safe 359, 1190
  scaling of variables 381

singular Jacobian 386
solving stiff ODEs 740
for square root of number 912, 1356
Niederreiter sequence 300
NL2SOL 683
Noise
bursty 889
effect on maximum entropy method 567
equivalent bandwidth 548
fitting data which contains 647f., 650
model, for optimal filtering 541
Nominal variable (statistics) 623
Nonexpansive projection operator 805
Non-interfering directions *see* Conjugate directions
Nonlinear eigenvalue problems 455
Nonlinear elliptic equations, multigrid method 874ff., 1339ff.
Nonlinear equations, in MEM inverse problems 813
Nonlinear equations, roots of 340ff.
Nonlinear instability 831
Nonlinear integral equations 781, 787
Nonlinear programming 436
Nonnegativity constraints 423
Nonparametric statistics 633ff., 1277ff.
Nonpolynomial complete (NP-complete) 438
Norm, of matrix 50
Normal (Gaussian) distribution 267, 652, 682, 798, 1294
central limit theorem 652f.
deviates from 279f., 571, 1152
kurtosis of 607
multivariate 690
semi-invariants of 608
tails compared to Poisson 653
two-dimensional (binormal) 631
variance of skewness of 606
Normal equations (fitting) 26, 645, 666ff., 795, 800, 1288
often are singular 670
Normalization
of Bessel functions 175
of floating-point representation 19, 882, 1343
of functions 142, 765
of modified Bessel functions 232
not() intrinsic function 951
Notch filter 551, 555f.
NP-complete problem 438
nr.f90 (module file) 936, 1362, 1384ff.
nrerror() utility function 989, 995
nrtype.f90 (module file) 936f.
named constants 1361
nrutil.f90 (module file) 936, 1070, 1362, 1364ff.
table of contents 1364
Null hypothesis 603
nullify statement 953f., 1070, 1302
Nullity 53
Nullspace 25, 53f., 449, 795
Number-theoretic transforms 503f.
Numeric inquiry functions 2/xiv, 1107, 1231, 1343
Numerical derivatives 180ff., 645, 1075
Numerical integration *see* Quadrature
Numerical intrinsic functions 946, 951f.
Numerical Recipes
compatibility with First Edition 4
Example Book 3
Fortran 90 types 936f., 1361
how to get programs 1/xx, 2/xvii
how to report bugs 1/iv, 2/iv
interface blocks (Fortran 90) 937, 941f., 1084, 1384ff.
no warranty on 1/xx, 2/xvii
plan of two-volume edition 1/xiii
table of dependencies 921ff., 1434ff.
as trademark 1/xxiii, 2/xx
utility functions (Fortran 90) 936f., 945, 968, 970, 972ff., 977, 984, 987ff., 1015, 1071f., 1361ff.
Numerical Recipes Software 1/xv, 1/xxiiff., 2/xviiff.
address and fax number 1/iv, 1/xxii, 2/iv, 2/xix
Web home page 1/xx, 2/xvii
Nyquist frequency 494ff., 520, 543, 545, 569ff.
Nystrom method 782f., 789, 1325
product version 789, 1331

Object extensibility 8
Objective function 424
Object-oriented programming 2/xvi, 2, 8
Oblateness parameter 764
Obsolete features *see* Fortran, Obsolescent features
Octal constant, initialization 959
Odd-even ordering
allows parallelization 1333
in Gauss-Seidel relaxation 866, 869, 1338
in successive over-relaxation (SOR) 859, 1332
Odd parity 888
OEM information 1/xxii
One-sided power spectral density 492
ONLY option, for USE statement 941, 957, 1067
Operation count
balancing 476
Bessel function evaluation 228
bisection method 346
Cholesky decomposition 90
coefficients of interpolating polynomial 114f.
complex multiplication 97
cubic spline interpolation 109
evaluating polynomial 168
fast Fourier transform (FFT) 498
Gauss-Jordan elimination 34, 39
Gaussian elimination 34
Givens reduction 463
Householder reduction 467
interpolation 100
inverse iteration 488
iterative improvement 48
Jacobi transformation 460
Kendall's tau 637

linear congruential generator 268
LU decomposition 36, 39
matrix inversion 97
matrix multiplication 96
maximum entropy method 567
multidimensional minimization 413f.
multigrid method 862
multiplication 909
polynomial evaluation 97f., 168
QL method 470, 473
QR decomposition 92
QR method for Hessenberg matrices 484
reduction to Hessenberg form 479
selection by partitioning 333
sorting 320ff.
Spearman rank-order coefficient 638
Toeplitz matrix 83
Vandermonde matrix 83
Operator overloading 2/xiif., 7
Operator splitting 823, 847f., 861
Operator, user-defined 2/xii
Optimal feasible vector 424
Optimal (Wiener) filtering 535, 539ff., 558, 644
compared with regularization 801
Optimization *see* Minimization
Optimization of code 2/xiii
Optional argument 2/xiv, 947f., 1092, 1228, 1230, 1256, 1272, 1275, 1340
dim 948
mask 948, 968, 1038
testing for 952
Ordering Numerical Recipes 1/xxf., 2/xviif.
Ordinal variable (statistics) 623
Ordinary differential equations *see* Differential equations
Orthogonal *see* Orthonormal functions; Orthonormal polynomials
Orthogonal transformation 452, 463ff., 469, 584, 1227
Orthonormal basis, constructing 58, 94, 1039
Orthonormal functions 142, 246
Orthonormal polynomials
Chebyshev 144, 184ff., 1076ff.
construct for arbitrary weight 151ff., 1064
in Gauss-Hermite integration 147, 1062
and Gaussian quadrature 142, 1009, 1061
Gaussian weights from recurrence 150, 1064
Hermite 144, 1062
Jacobi 144, 1063
Laguerre 144, 1060
Legendre 144, 1059
weight function $\log x$ 153
Orthonormality 51, 142, 463
Outer product Gaussian elimination 1017
Outer product of matrices (denoted by cross) 66, 420, 949, 969f., 989, 1000ff., 1017, 1026, 1040, 1076, 1200, 1216, 1275
outerand() utility function 989, 1002, 1015
outerdiff() utility function 989, 1001
outerdiv() utility function 989, 1001
outerprod() utility function 970, 989, 1000, 1017, 1026, 1040, 1076, 1200, 1216, 1275
outersum() utility function 989, 1001
Outgoing wave boundary conditions 820
Outlier 605, 653, 656, 694, 697
see also Robust estimation
Overcorrection 857
Overflow 882, 1343
how to avoid in modulo multiplication 269
in complex arithmetic 171
Overlap-add and overlap-save methods 536f.
Overloading
operator 2/xiif.
procedures 940, 1015, 1083, 1094, 1096
Overrelaxation parameter 857, 1332
choice of 858

Pack() intrinsic function 945, 950, 964, 991, 1031
communication bottleneck 969
for index table 1176
for partition-exchange 1170
for selection 1178
for selective evaluation 1087
Pack-unpack idiom 1087, 1134, 1153
Padé approximant 194ff., 1080f.
Padé approximation 105
Parabolic interpolation 395, 1204
Parabolic partial differential equations 818, 838ff.
Parallel axis theorem 308
Parallel programming 2/xv, 941, 958ff., 962ff., 965f., 968f., 987
array operations 964f.
array ranking 1278f.
band diagonal linear equations 1021
Bessel functions 1107ff.
broadcasts 965ff.
C and C++ 2/viii
communication costs 969, 981, 1250
counting do-loops 1015
cyclic reduction 974
deflation 977ff.
design matrix 1082
dimensional expansion 965ff.
eigensystems 1226, 1229f.
fast Fourier transform (FFT) 981, 1235ff., 1250
in Fortran 90 963ff.
Fortran 90 tricks 1009, 1274, 1278, 1280
function evaluation 986, 1009, 1084f., 1087, 1090, 1102, 1128, 1134
Gaussian quadrature 1009, 1061
geometric progressions 972
index loss 967f., 1038
index table 1176f.
interprocessor communication 981
Kendall's tau 1280
linear algebra 969f., 1000ff., 1018f., 1026, 1040, 1200, 1326
linear recurrence 973f., 1073ff.
logo 2/viii, 1009
masks 967f., 1006f., 1038, 1102, 1200, 1226, 1305, 1333f., 1368, 1378, 1382
merge statement 1010

MIMD (multiple instruction, multiple data) 964, 985f., 1084
MMP (massively multiprocessor) machines 965ff., 974, 984, 1016ff., 1226ff., 1250
nrutil.f90 (module file) 1364ff.
odd-even ordering 1333
one-dimensional FFT 982f.
parallel note icon 1009
partial differential equations 1333
in-place selection 1178f.
polynomial coefficients from roots 980
polynomial evaluation 972f., 977, 998
random numbers 1009, 1141ff.
recursive doubling 973f., 976f., 979, 988, 999, 1071ff.
scatter-with-combine 984, 1002f., 1032f.
second order recurrence 974f., 1074
SIMD (Single Instruction Multiple Data) 964, 985f., 1009, 1084f.
singular value decomposition (SVD) 1026
sorting 1167ff., 1171, 1176f.
special functions 1009
SSP (small-scale parallel) machines 965ff., 984, 1010ff., 1016ff., 1059f., 1226ff., 1250
subvector scaling 972, 974, 996, 1000
successive over-relaxation (SOR) 1333
supercomputers 2/viii, 962
SVD algorithm 1026
synthetic division 977ff., 999, 1048, 1071f., 1079, 1192
tridiagonal systems 975f., 1018, 1229f.
utilities 1364ff.
vector reduction 972f., 977, 998
vs. serial programming 965, 987
PARAMETER attribute 1012
Parameters in fitting function 651, 684ff.
Parity bit 888
Park and Miller minimal standard random generator 269, 1142
Parkinson's Law 328
Parseval's Theorem 492, 544
discrete form 498
Partial differential equations 818ff., 1332ff.
advective equation 826
alternating-direction implicit method (ADI) 847, 861f.
amplification factor 828, 834
analyze/factorize/operate package 824
artificial viscosity 831, 837
biconjugate gradient method 824
boundary conditions 819ff.
boundary value problems 819, 848
Cauchy problem 818f.
caution on high-order methods 844f.
Cayley's form 844
characteristics 818
Chebyshev acceleration 859f., 1332
classification of 818f.
comparison of rapid methods 854
conjugate gradient method 824
Courant condition 829, 832ff., 836
Courant condition (multidimensional) 846
Crank-Nicholson method 840, 842, 844, 846
cyclic reduction (CR) method 848f., 852ff.
diffusion equation 818, 838ff., 846, 855
Dirichlet boundary conditions 508, 820, 840, 850, 856, 858
elliptic, defined 818
error, varieties of 831ff.
explicit vs. implicit differencing 827
FACR method 854
finite difference method 821ff.
finite element methods 824
flux-conservative initial value problems 825ff.
forward Euler differencing 826f.
Forward Time Centered Space (FTCS) 827ff., 839ff., 843, 855
Fourier analysis and cyclic reduction (FACR) 848ff., 854
Gauss-Seidel method (relaxation) 855, 864ff., 876, 1338, 1341
Godunov's method 837
Helmholtz equation 852
hyperbolic 818, 825f.
implicit differencing 840
incomplete Cholesky conjugate gradient method (ICCG) 824
inhomogeneous boundary conditions 850f.
initial value problems 818f.
initial value problems, recommendations on 838ff.
Jacobi's method (relaxation) 855ff., 864
Laplace's equation 818
Lax method 828ff., 836, 845f.
Lax method (multidimensional) 845f.
matrix methods 824
mesh-drift instability 834f.
Monte Carlo methods 824
multidimensional initial value problems 844ff.
multigrid method 824, 862ff., 1009, 1334ff.
Neumann boundary conditions 508, 820, 840, 851, 858
nonlinear diffusion equation 842
nonlinear instability 831
numerical dissipation or viscosity 830
operator splitting 823, 847f., 861
outgoing wave boundary conditions 820
parabolic 818, 838ff.
parallel computing 1333
periodic boundary conditions 850, 858
piecewise parabolic method (PPM) 837
Poisson equation 818, 852
rapid (Fourier) methods 508ff., 824, 848ff.
relaxation methods 823, 854ff., 1332f.
Schrödinger equation 842ff.
second-order accuracy 833ff., 840
shock 831, 837
sparse matrices from 64
spectral methods 825
spectral radius 856ff., 862
stability vs. accuracy 830
stability vs. efficiency 821
staggered grids 513, 852
staggered leapfrog method 833f.
strongly implicit procedure 824

successive over-relaxation (SOR) 857ff., 862, 866, 1332f.
time splitting 847f., 861
two-step Lax-Wendroff method 835ff.
upwind differencing 832f., 837
variational methods 824
varieties of error 831ff.
von Neumann stability analysis 827f., 830, 833f., 840
wave equation 818, 825f.
see also Elliptic partial differential equations; Finite difference equations (FDEs)
Partial pivoting 29
Partition-exchange 323, 333
and pack() intrinsic function 1170
Partitioned matrix, inverse of 70
Party tricks 95ff., 168
Parzen window 547
Pascal, Numerical Recipes in 2/x, 2/xvii, 1
Pass-the-buck idiom 1102, 1128
Path integration, for function evaluation 201ff., 263, 1138
Pattern multiply of sparse matrices 74
PBCG (preconditioned biconjugate gradient method) 78f., 824
PC methods *see* Predictor-corrector methods
PCGPACK 71
PDEs *see* Partial differential equations
Pearson's r 630ff., 1276
PECE method 741
Pentagon, symmetries of 895
Percentile 320
Period of linear congruential generator 268
Periodic boundary conditions 850, 858
Periodogram 543ff., 566, 1258ff.
Lomb's normalized 569f., 574f., 1258ff.
variance of 544f.
Perl (programming language) 1/xvi
Perron's theorems, for convergence of recurrence relations 174f.
Perturbation methods for matrix inversion 65ff.
Phase error 831
Phase-locked loop 700
Phi statistic 625
Phillips-Twomey method 799ff.
Pi, computation of 906ff., 1352ff., 1357f.
Piecewise parabolic method (PPM) 837
Pincherle's theorem 175
Pivot element 29, 33, 757
in linear programming 428f.
Pivoting 27, 29ff., 46, 66, 90, 1014
full 29, 1014
implicit 30, 38, 1014, 1017
in LU decomposition 37f., 1017
partial 29, 33, 37f., 1017
and QR decomposition 92
in reduction to Hessenberg form 478
in relaxation method 757
as row and column operations 32
for tridiagonal systems 43
Pixel 519, 596, 803, 811
PL/1 2/x
Planck's constant 842
Plane rotation *see* Givens reduction; Jacobi transformation (or rotation)
Platykurtic distribution 606
Plotting of functions 342, 1182f.
POCS (projection onto convex sets) 805
Poetry 5f.
Pointer (Fortran 90) 2/xiiif., 938f., 944f., 953ff., 1197, 1212, 1266
as alias 939, 944f., 1286, 1333
allocating an array 941
allocating storage for derived type 955
for array of arrays 956, 1336
array of, forbidden 956, 1337
associated with target 938f., 944f., 952f., 1197
in Fortran 95 961
to function, forbidden 1067, 1210
initialization to null 2/xv, 961
returning array of unknown size 955f., 1184, 1259, 1261, 1327
undefined status 952f., 961, 1070, 1266, 1302
Poisson equation 519, 818, 852
Poisson probability function
cumulative 214
deviates from 281, 283ff., 571, 1154
semi-invariants of 608
tails compared to Gaussian 653
Poisson process 278, 282ff., 1153
Polak-Ribiere algorithm 390, 414ff., 1214
Poles *see* Complex plane, poles in
Polishing of roots 356, 363ff., 370f., 1193
poly() utility function 973, 977, 989, 998, 1072, 1096, 1192, 1258, 1284
Polymorphism 8
Polynomial interpolation 99, 102ff., 1043
Aitken's algorithm 102
in Bulirsch-Stoer method 724, 726, 1305
coefficients for 113ff., 1047f.
Lagrange's formula 84, 102f.
multidimensional 116ff., 1049ff.
Neville's algorithm 102f., 105, 134, 182, 1043
pathology in determining coefficients for 116
in predictor-corrector method 740
smoothing filters 645
see also Interpolation
Polynomials 167ff.
algebraic manipulations 169, 1072
approximate roots of Hermite polynomials 1062
approximate roots of Jacobi polynomials 1064
approximate roots of Laguerre polynomials 1061
approximating modified Bessel functions 230
approximation from Chebyshev coefficients 191, 1078f.
AUTODIN-II 890
CCITT 889f.
characteristic 368, 1193
characteristic, for digital filters 554, 559, 1257

characteristic, for eigenvalues of matrix 449, 469
Chebyshev 184ff., 1076ff.
coefficients from roots 980
CRC-16 890
cumulants of 977, 999, 1071f., 1192, 1365, 1378f.
deflation 362ff., 370f., 977
derivatives of 167, 978, 1071
division 84, 169, 362, 370, 977, 1072
evaluation of 167, 972, 977, 998f., 1071, 1258, 1365, 1376ff.
evaluation of derivatives 167, 978, 1071
extrapolation in Bulirsch-Stoer method 724, 726, 1305f.
extrapolation in Romberg integration 134
fitting 83, 114, 191, 645, 665, 674, 1078f., 1291
generator for CRC 889
ill-conditioned 362
masked evaluation of 1378
matrix method for roots 368, 1193
minimax 186, 198, 1076
monic 142f.
multiplication 169
operation count for 168
orthonormal 142, 184, 1009, 1061
parallel operations on 977ff., 998f., 1071f., 1192
primitive modulo 2 287ff., 301f., 889
roots of 178ff., 362ff., 368, 1191ff.
shifting of 192f., 978, 1079
stopping criterion in root finding 366
poly_term() utility function 974, 977, 989, 999, 1071f., 1192
Port, serial data 892
Portability 3, 963
Portable random number generator *see* Random number generator
Positive definite matrix, testing for 90
Positivity constraints 423
Postal Service (U.S.), barcode 894
PostScript 1/xvi, 1/xxiii, 2/xx
Powell's method 389, 402, 406ff., 1210ff.
Power (in a signal) 492f.
Power series 159ff., 167, 195
economization of 192f., 1061, 1080
Padé approximant of 194ff., 1080f.
Power spectral density *see* Fourier transform; Spectral density
Power spectrum estimation *see* Fourier transform; Spectral density
PowerStation, Microsoft Fortran 2/xix
PPM (piecewise parabolic method) 837
Precision
converting to double 1362
floating point 882, 937, 1343, 1361ff.
multiple 906ff., 1352ff., 1362
Preconditioned biconjugate gradient method (PBCG) 78f.
Preconditioning, in conjugate gradient methods 824
Predictor-corrector methods 702, 730, 740ff.
Adams-Bashforth-Moulton schemes 741
adaptive order methods 744
compared to other methods 740
fallacy of multiple correction 741
with fixed number of iterations 741
functional iteration vs. Newton's rule 742
multivalue compared with multistep 742ff.
starting and stopping 742, 744
stepsize control 742f.
present() intrinsic function 952
Prime numbers 915
Primitive polynomials modulo 2 287ff., 301f., 889
Principal directions 408f., 1210
Principal solution, of inverse problem 797
PRIVATE attribute 957, 1067
Prize, $1000 offered 272, 1141, 1150f.
Probability *see* Random number generator; Statistical tests
Probability density, change of variables in 278f.
Procedure *see* Program(s); Subprogram
Process loss 548
product() intrinsic function 948
Product Nystrom method 789, 1331
Program(s)
as black boxes 1/xviii, 6, 26, 52, 205, 341, 406
dependencies 921ff., 1434ff.
encapsulation 7
interfaces 2, 8
modularization 7f.
organization 5ff.
type declarations 2
typography of 2f., 12, 937
validation 3f.
Programming, serial vs. parallel 965, 987
Projection onto convex sets (POCS) 805
Projection operator, nonexpansive 805
Prolongation operator 864, 1337
Protocol, for communications 888
PSD (power spectral density) *see* Fourier transform; Spectral density
Pseudo-random numbers 266ff., 1141ff.
PUBLIC attribute 957, 1067
Puns, particularly bad 167, 744, 747
PURE attribute 2/xv, 960f., 964, 986
put_diag() utility function 985, 990, 1005, 1200
Pyramidal algorithm 586, 1264
Pythagoreans 392

QL *see* Eigensystems
QR *see* Eigensystems
QR decomposition 91f., 382, 386, 1039f., 1199
backsubstitution 92, 1040
and least squares 668
operation count 92
pivoting 92
updating 94, 382, 386, 1041, 1199
use for orthonormal basis 58, 94
Quadratic
convergence 49, 256, 351, 356, 409f., 419, 906
equations 20, 178, 391, 457

interpolation 353, 364
programming 436
Quadrature 123ff., 1052ff.
adaptive 123, 190, 788
alternative extended Simpson's rule 128
arbitrary weight function 151ff., 789, 1064, 1328
automatic 154
Bode's rule 126
change of variable in 137ff., 788, 1056ff.
by Chebyshev fitting 124, 189, 1078
classical formulas for 124ff.
Clenshaw-Curtis 124, 190, 512f.
closed formulas 125, 127f.
and computer science 881
by cubic splines 124
error estimate in solution 784
extended midpoint rule 129f., 135, 1054f.
extended rules 127ff., 134f., 786, 788ff., 1326, 1328
extended Simpson's rule 128
Fourier integrals 577ff., 1261ff.
Fourier integrals, infinite range 583
Gauss-Chebyshev 144, 512f.
Gauss-Hermite 144, 789, 1062
Gauss-Jacobi 144, 1063
Gauss-Kronrod 154
Gauss-Laguerre 144, 789, 1060
Gauss-Legendre 144, 783, 789, 1059, 1325
Gauss-Lobatto 154, 190, 512
Gauss-Radau 154
Gaussian integration 127, 140ff., 781, 783, 788f., 1009, 1059ff., 1325, 1328f.
Gaussian integration, nonclassical weight function 151ff., 788f., 1064f., 1328f.
for improper integrals 135ff., 789, 1055, 1328
for integral equations 781f., 786, 1325ff.
Monte Carlo 124, 155ff., 295ff., 306ff., 1161ff.
multidimensional 124, 155ff., 1052, 1065ff.
multidimensional, by recursion 1052, 1065
Newton-Cotes formulas 125ff., 140
open formulas 125ff., 129f., 135
related to differential equations 123
related to predictor-corrector methods 740
Romberg integration 124, 134f., 137, 182, 717, 788, 1054f., 1065, 1067
semi-open formulas 130
Simpson's rule 126, 133, 136f., 583, 782, 788ff., 1053
Simpson's three-eighths rule 126, 789f.
singularity removal 137ff., 788, 1057ff., 1328ff.
singularity removal, worked example 792, 1328ff.
trapezoidal rule 125, 127, 130ff., 134f., 579, 583, 782, 786, 1052ff., 1326f.
using FFTs 124
weight function $\log x$ 153
see also Integration of functions
Quadrature mirror filter 585, 593
Quantum mechanics, Uncertainty Principle 600
Quartile value 320
Quasi-Newton methods for minimization 390, 418ff., 1215
Quasi-random sequence 299ff., 318, 881, 888
Halton's 300
for Monte Carlo integration 304, 309, 318
Sobol's 300ff., 1160
see also Random number generator
Quicksort 320, 323ff., 330, 333, 1169f.
Quotient-difference algorithm 164

R-estimates 694
Radioactive decay 278
Radix base for floating point arithmetic 476, 882, 907, 913, 1231, 1343, 1357
Radix conversion 902, 906, 913, 1357
radix() intrinsic function 1231
Radix sort 1172
Ramanujan's identity for π 915
Random bits, generation of 287ff., 1159f.
Random deviates 266ff., 1141ff.
binomial 285f., 1155
exponential 278, 1151f.
gamma distribution 282f., 1153
Gaussian 267, 279f., 571, 798, 1152f.
normal 267, 279f., 571, 1152f.
Poisson 283ff., 571, 1154f.
quasi-random sequences 299ff., 881, 888, 1160f.
uniform 267ff., 1158f., 1166
uniform integer 270, 274ff.
Random number generator 266ff., 1141ff.
bitwise operations 287
Box-Muller algorithm 279, 1152
Data Encryption Standard 290ff., 1144, 1156ff.
good choices for modulus, multiplier and increment 274ff.
initializing 1144ff.
for integer-valued probability distribution 283f., 1154
integer vs. real implementation 273
L'Ecuyer's long period 271f.
lagged Fibonacci generator 1142, 1148ff.
linear congruential generator 267ff., 1142
machine language 269
Marsaglia shift register 1142, 1148ff.
Minimal Standard, Park and Miller's 269, 1142
nonrandomness of low-order bits 268f.
parallel 1009
perfect 272, 1141, 1150f.
planes, numbers lie on 268
portable 269ff., 1142
primitive polynomials modulo 2 287ff.
pseudo-DES 291, 1144, 1156ff.
quasi-random sequences 299ff., 881, 888, 1160f.
quick and dirty 274
quicker and dirtier 275
in Quicksort 324
random access to nth number 293

random bits 287ff., 1159f.
recommendations 276f.
rejection method 281ff.
serial 1141f.
shuffling procedure 270, 272
in simulated annealing method 438
spectral test 274
state space 1143f.
state space exhaustion 1141
subtractive method 273, 1143
system-supplied 267f.
timings 276f., 1151
transformation method 277ff.
trick for trigonometric functions 280
Random numbers *see* Monte Carlo; Random deviates
Random walk 20
random_number() intrinsic function 1141, 1143
random_seed() intrinsic function 1141
RANDU, infamous routine 268
Range 53f.
Rank (matrix) 53
kernel of finite 785
Rank (sorting) 320, 332, 1176
Rank (statistics) 633ff., 694f., 1277
Kendall's tau 637ff., 1279
Spearman correlation coefficient 634f., 1277ff.
sum squared differences of 634, 1277
Ratio variable (statistics) 623
Rational Chebyshev approximation 197ff., 1081f.
Rational function 99, 167ff., 194ff., 1080f.
approximation for Bessel functions 225
approximation for continued fraction 164, 211, 219f.
Chebyshev approximation 197ff., 1081f.
evaluation of 170, 1072f.
extrapolation in Bulirsch-Stoer method 718ff., 726, 1306f.
interpolation and extrapolation using 99, 104ff., 194ff., 718ff., 726
as power spectrum estimate 566
interpolation and extrapolation using 1043f., 1080ff., 1306
minimax 198
Re-entrant procedure 1052
real() intrinsic function, ambiguity of 947
Realizable (causal) 552, 554f.
reallocate() utility function 955, 990, 992, 1070, 1302
Rearranging *see* Sorting
Reciprocal, multiple precision 910f., 1355f.
Record, in data file 329
Recurrence relation 172ff., 971ff.
arithmetic progression 971f., 996
associated Legendre polynomials 247
Bessel function 172, 224, 227f., 234
binomial coefficients 209
Bulirsch-Stoer 105f.
characteristic polynomial of tridiagonal matrix 469
Clenshaw's recurrence formula 176f.
and continued fraction 175
continued fraction evaluation 164f.
convergence 175
cosine function 172, 500
cyclic reduction 974
dominant solution 174
exponential integrals 172
gamma function 206
generation of random bits 287f.
geometric progression 972, 996
Golden Mean 21
Legendre polynomials 172
minimal vs. dominant solution 174
modified Bessel function 232
Neville's 103, 182
orthonormal polynomials 142
Perron's theorems 174f.
Pincherle's theorem 175
for polynomial cumulants 977, 999, 1071f.
polynomial interpolation 103, 183
primitive polynomials modulo 2 287f.
random number generator 268
rational function interpolation 105f., 1043
recursive doubling 973, 977, 988, 999, 1071f., 1073
second order 974f., 1074
sequence of trig functions 173
sine function 172, 500
spherical harmonics 247
stability of 21, 173ff., 177, 224f., 227f., 232, 247, 975
trig functions 572
weight of Gaussian quadrature 144f.
Recursion
in Fortran 90 958
in multigrid method 865, 1009, 1336
Recursive doubling 973f., 979
cumulants of polynomial 977, 999, 1071f.
linear recurrences 973, 988, 1073
tridiagonal systems 976
RECURSIVE keyword 958, 1065, 1067
Recursive Monte Carlo integration 306ff., 1161
Recursive procedure 2/xiv, 958, 1065, 1067, 1166
as parallelization tool 958
base case 958
for multigrid method 1009, 1336
re-entrant 1052
Recursive stratified sampling 314ff., 1164ff.
Red-black *see* Odd-even ordering
Reduction functions 948ff.
Reduction of variance in Monte Carlo integration 299, 306ff.
References (explanation) 4f.
References (general bibliography) 916ff., 1359f.
Reflection formula for gamma function 206
Regula falsi (false position) 347ff., 1185f.
Regularity condition 775
Regularization
compared with optimal filtering 801
constrained linear inversion method 799ff.
of inverse problems 796ff.
linear 799ff.
nonlinear 813

objective criterion 802
Phillips-Twomey method 799ff.
Tikhonov-Miller 799ff.
trade-off curve 799
two-dimensional 803
zeroth order 797
see also Inverse problems
Regularizing operator 798
Reid, John 2/xiv, 2/xvi
Rejection method for random number generator 281ff.
Relaxation method
for algebraically difficult sets 763
automated allocation of mesh points 774f., 777
computation of spheroidal harmonics 764ff., 1319ff.
for differential equations 746f., 753ff., 1316ff.
elliptic partial differential equations 823, 854ff., 1332f.
example 764ff., 1319ff.
Gauss-Seidel method 855, 864ff., 876, 1338, 1341
internal boundary conditions 775ff.
internal singular points 775ff.
Jacobi's method 855f., 864
successive over-relaxation (SOR) 857ff., 862, 866, 1332f.
see also Multigrid method
Remes algorithms
exchange algorithm 553
for minimax rational function 199
reshape() intrinsic function 950
communication bottleneck 969
order keyword 1050, 1246
Residual 49, 54, 78
in multigrid method 863, 1338
Resolution function, in Backus-Gilbert method 807
Response function 531
Restriction operator 864, 1337
RESULT keyword 958, 1073
Reward, $1000 offered 272, 1141, 1150f.
Richardson's deferred approach to the limit 134, 137, 182, 702, 718ff., 726, 788, 869
see also Bulirsch-Stoer method
Richtmyer artificial viscosity 837
Ridders' method, for numerical derivatives 182, 1075
Ridders' method, root finding 341, 349, 351, 1187
Riemann shock problem 837
Right eigenvalues and eigenvectors 451
Rise/fall time 548f.
Robust estimation 653, 694ff., 700, 1294
Andrew's sine 697
average deviation 605
double exponential errors 696
Kalman filtering 700
Lorentzian errors 696f.
mean absolute deviation 605
nonparametric correlation 633ff., 1277
Tukey's biweight 697
use of a priori covariances 700
see also Statistical tests
Romberg integration 124, 134f., 137, 182, 717, 788, 1054f., 1065
Root finding 143, 340ff., 1009, 1059
advanced implementations of Newton's rule 386
Bairstow's method 364, 370, 1193
bisection 343, 346f., 352f., 359, 390, 469, 698, 1184f.
bracketing of roots 341, 343ff., 353f., 362, 364, 369, 1183f.
Brent's method 341, 349, 660f., 1188f., 1286
Broyden's method 373, 382f., 386, 1199
compared with multidimensional minimization 375
complex analytic functions 364
in complex plane 204
convergence criteria 347, 374
deflation of polynomials 362ff., 370f., 1192
without derivatives 354
double root 341
eigenvalue methods 368, 1193
false position 347ff., 1185f.
Jenkins-Traub method 369
Laguerre's method 341, 366f., 1191f.
Lehmer-Schur algorithm 369
Maehly's procedure 364, 371
matrix method 368, 1193
Muller's method 364, 372
multiple roots 341
Newton's rule 143f., 180, 341, 355ff., 362, 364, 370, 372ff., 376, 469, 740, 749f., 754, 787, 874, 876, 911f., 1059, 1189, 1194, 1196, 1314ff., 1339, 1341, 1355f.
pathological cases 343, 356, 362, 372
polynomials 341, 362ff., 449, 1191f.
in relaxation method 754, 1316
Ridders' method 341, 349, 351, 1187
root-polishing 356, 363ff., 369ff., 1193
safe Newton's rule 359, 1190
secant method 347ff., 358, 364, 399, 1186f.
in shooting method 746, 749f., 1314f.
singular Jacobian in Newton's rule 386
stopping criterion for polynomials 366
use of minimum finding 341
using derivatives 355ff., 1189
zero suppression 372
see also Roots
Root polishing 356, 363ff., 369ff., 1193
Roots
Chebyshev polynomials 184
complex *n*th root of unity 999f., 1379
cubic equations 179f.
Hermite polynomials, approximate 1062
Jacobi polynomials, approximate 1064
Laguerre polynomials, approximate 1061
multiple 341, 364ff., 1192
nonlinear equations 340ff.
polynomials 341, 362ff., 449, 1191f.
quadratic equations 178

reflection in unit circle 560, 1257
square, multiple precision 912, 1356
see also Root finding
Rosenbrock method 730, 1308
compared with semi-implicit extrapolation 739
stepsize control 731, 1308f.
Roundoff error 20, 881, 1362
bracketing a minimum 399
compile time vs. run time 1012
conjugate gradient method 824
eigensystems 458, 467, 470, 473, 476, 479, 483
extended trapezoidal rule 132
general linear least squares 668, 672
graceful 883, 1343
hardware aspects 882, 1343
Householder reduction 466
IEEE standard 882f., 1343
interpolation 100
least squares fitting 658, 668
Levenberg-Marquardt method 679
linear algebraic equations 23, 27, 29, 47, 56, 84, 1022
linear predictive coding (LPC) 564
magnification of 20, 47, 1022
maximum entropy method (MEM) 567
measuring 881f., 1343
multidimensional minimization 418, 422
multiple roots 362
numerical derivatives 180f.
recurrence relations 173
reduction to Hessenberg form 479
series 164f.
straight line fitting 658
variance 607
Row degeneracy 22
Row-indexed sparse storage 71f., 1030
transpose 73f.
Row operations on matrix 28, 31f.
Row totals 624
RSS algorithm 314ff., 1164
RST properties (reflexive, symmetric, transitive) 338
Runge-Kutta method 702, 704ff., 731, 740, 1297ff., 1308
Cash-Karp parameters 710, 1299f.
embedded 709f., 731, 1298, 1308
high-order 705
quality control 722
stepsize control 708ff.
Run-length encoding 901
Runge-Kutta method
high-order 1297
stepsize control 1298f.
Rybicki, G.B. 84ff., 114, 145, 252, 522, 574, 600

S-box for Data Encryption Standard 1148
Sampling
importance 306f.
Latin square or hypercube 305f.
recursive stratified 314ff., 1164
stratified 308f.
uneven or irregular 569, 648f., 1258
Sampling theorem 495, 543
for numerical approximation 600ff.
Sande-Tukey FFT algorithm 503
SAVE attribute 953f., 958f., 961, 1052, 1070, 1266, 1293
redundant use of 958f.
SAVE statements 3
Savitzky-Golay filters
for data smoothing 644ff., 1283f.
for numerical derivatives 183, 645
scale() intrinsic function 1107
Scallop loss 548
Scatter-with-combine functions 984, 1002f., 1032, 1366, 1380f.
scatter_add() utility function 984, 990, 1002, 1032
scatter_max() utility function 984, 990, 1003
Schonfelder, Lawrie 2/xi
Schrage's algorithm 269
Schrödinger equation 842ff.
Schultz's method for matrix inverse 49, 598
Scope 956ff., 1209, 1293, 1296
Scoping unit 939
SDLC checksum 890
Searching
with correlated values 111, 1046f.
an ordered table 110f., 1045f.
selection 333, 1177f.
Secant method 341, 347ff., 358, 364, 399, 1186f.
Broyden's method 382f., 1199f.
multidimensional (Broyden's) 373, 382f., 1199
Second Euler-Maclaurin summation formula 135f.
Second order differential equations 726, 1307
Seed of random number generator 267, 1146f.
select case statement 2/xiv, 1010, 1036
Selection 320, 333, 1177f.
find m largest elements 336, 1179f.
heap algorithm 336, 1179
for median 698, 1294
operation count 333
by packing 1178
parallel algorithms 1178
by partition-exchange 333, 1177f.
without rearrangement 335, 1178f.
timings 336
use to find median 609
Semi-implicit Euler method 730, 735f.
Semi-implicit extrapolation method 730, 735f., 1310f.
compared with Rosenbrock method 739
stepsize control 737, 1311f.
Semi-implicit midpoint rule 735f., 1310f.
Semi-invariants of a distribution 608
Sentinel, in Quicksort 324, 333
Separable kernel 785
Separation of variables 246
Serial computing
convergence of quadrature 1060
random numbers 1141
sorting 1167
Serial data port 892

Series 159ff.
accelerating convergence of 159ff.
alternating 160f., 1070
asymptotic 161
Bessel function K_ν 241
Bessel function Y_ν 235
Bessel functions 160, 223
cosine integral 250
divergent 161
economization 192f., 195, 1080
Euler's transformation 160f., 1070
exponential integral 216, 218
Fresnel integral 248
hypergeometric 202, 263, 1138
incomplete beta function 219
incomplete gamma function 210, 1090f.
Laurent 566
relation to continued fractions 163f.
roundoff error in 164f.
sine and cosine integrals 250
sine function 160
Taylor 355f., 408, 702, 709, 754, 759
transformation of 160ff., 1070
van Wijngaarden's algorithm 161, 1070
Shaft encoder 886
Shakespeare 9
Shampine's Rosenbrock parameters 732, 1308
shape() intrinsic function 938, 949
Shell algorithm (Shell's sort) 321ff., 1168
Sherman-Morrison formula 65ff., 83, 382
Shifting of eigenvalues 449, 470f., 480
Shock wave 831, 837
Shooting method
computation of spheroidal harmonics 772, 1321ff.
for differential equations 746, 749ff., 770ff., 1314ff., 1321ff.
for difficult cases 753, 1315f.
example 770ff., 1321ff.
interior fitting point 752, 1315f., 1323ff.
Shuffling to improve random number generator 270, 272
Side effects
prevented by data hiding 957, 1209, 1293, 1296
and PURE subprograms 960
Sidelobe fall-off 548
Sidelobe level 548
sign() intrinsic function, modified in Fortran 95 961
Signal, bandwidth limited 495
Significance (numerical) 19
Significance (statistical) 609f.
one- vs. two-sided 632
peak in Lomb periodogram 570
of 2-d K-S test 640, 1281
two-tailed 613
SIMD machines (Single Instruction Multiple Data) 964, 985f., 1009, 1084f.
Similarity transform 452ff., 456, 476, 478, 482
Simplex
defined 402
method in linear programming 389, 402, 423ff., 431ff., 1216ff.
method of Nelder and Mead 389, 402ff., 444, 697f., 1208f., 1222ff.
use in simulated annealing 444, 1222ff.
Simpson's rule 124ff., 128, 133, 136f., 583, 782, 788f., 1053f.
Simpson's three-eighths rule 126, 789f.
Simulated annealing *see* Annealing, method of simulated
Simulation *see* Monte Carlo
Sine function
evaluated from $\tan(\theta/2)$ 173
recurrence 172
series 160
Sine integral 248, 250ff., 1123, 1125f.
continued fraction 250
series 250
see also Cosine integral
Sine transform *see* Fast Fourier transform (FFT); Fourier transform
Singleton's algorithm for FFT 525
Singular value decomposition (SVD) 23, 25, 51ff., 1022
approximation of matrices 58f.
backsubstitution 56, 1022f.
and bases for nullspace and range 53
confidence levels from 693f.
covariance matrix 693f.
fewer equations than unknowns 57
for inverse problems 797
and least squares 54ff., 199f., 668, 670ff., 1081, 1290f.
in minimization 410
more equations than unknowns 57f.
parallel algorithms 1026
and rational Chebyshev approximation 199f., 1081f.
of square matrix 53ff., 1023
use for ill-conditioned matrices 56, 58, 449
use for orthonormal basis 58, 94
Singularities
of hypergeometric function 203, 263
in integral equations 788ff., 1328
in integral equations, worked example 792, 1328ff.
in integrands 135ff., 788, 1055, 1328ff.
removal in numerical integration 137ff., 788, 1057ff., 1328ff.
Singularity, subtraction of the 789
SIPSOL 824
Six-step framework, for FFT 983, 1240
size() intrinsic function 938, 942, 945, 948
Skew array section 2/xii, 945, 960, 985, 1284
Skewness of distribution 606, 608, 1269
Smoothing
of data 114, 644ff., 1283f.
of data in integral equations 781
importance in multigrid method 865
sn function 261, 1137f.
Snyder, N.L. 1/xvi
Sobol's quasi-random sequence 300ff., 1160f.
Sonata 9
Sonnet 9
Sorting 320ff., 1167ff.
bubble sort 1168

bubble sort cautioned against 321
compared to selection 333
covariance matrix 669, 681, 1289
eigenvectors 461f., 1227
Heapsort 320, 327f., 336, 1171f., 1179
index table 320, 329f., 1170, 1173ff., 1176
operation count 320ff.
by packing 1171
parallel algorithms 1168, 1171f., 1176
Quicksort 320, 323ff., 330, 333, 1169f.
radix sort 1172
rank table 320, 332, 1176
ranking 329, 1176
by reshaping array slices 1168
Shell's method 321ff., 1168
straight insertion 321f., 461f., 1167, 1227
SP, defined 937
SPARC or SPARCstation 1/xxii, 2/xix, 4
Sparse linear equations 23, 63ff., 732, 1030
band diagonal 43, 1019ff.
biconjugate gradient method 77, 599, 1034
data type for 1030
indexed storage 71f., 1030
in inverse problems 804
minimum residual method 78
named patterns 64, 822
partial differential equations 822ff.
relaxation method for boundary value problems 754, 1316
row-indexed storage 71f., 1030
wavelet transform 584, 598
see also Matrix
Spearman rank-order coefficient 634f., 694f., 1277
Special functions *see* Function
Spectral analysis *see* Fourier transform; Periodogram
Spectral density 541
and data windowing 545ff.
figures of merit for data windows 548f.
normalization conventions 542f.
one-sided PSD 492
periodogram 543ff., 566, 1258ff.
power spectral density (PSD) 492f.
power spectral density per unit time 493
power spectrum estimation by FFT 542ff., 1254ff.
power spectrum estimation by MEM 565ff., 1258
two-sided PSD 493
variance reduction in spectral estimation 545
Spectral lines, how to smooth 644
Spectral methods for partial differential equations 825
Spectral radius 856ff., 862
Spectral test for random number generator 274
Spectrum *see* Fourier transform
Spherical Bessel functions 234
routine for 245, 1121
Spherical harmonics 246ff.
orthogonality 246
routine for 247f., 1122
stable recurrence for 247
table of 246
see also Associated Legendre polynomials
Spheroidal harmonics 764ff., 770ff., 1319ff.
boundary conditions 765
normalization 765
routine for 768ff., 1319ff., 1323ff.
Spline 100
cubic 107ff., 1044f.
gives tridiagonal system 109
natural 109, 1044f.
operation count 109
two-dimensional (bicubic) 120f., 1050f.
spread() intrinsic function 945, 950, 969, 1000, 1094, 1290f.
and dimensional expansion 966ff.
Spread matrix 808
Spread spectrum 290
Square root, complex 172
Square root, multiple precision 912, 1356f.
Square window 546, 1254ff.
SSP (small-scale parallel) machines 965ff., 972, 974, 984, 1011, 1016ff., 1021, 1059f., 1226ff., 1250
Stability 20f.
of Clenshaw's recurrence 177
Courant condition 829, 832ff., 836, 846
diffusion equation 840
of Gauss-Jordan elimination 27, 29
of implicit differencing 729, 840
mesh-drift in PDEs 834f.
nonlinear 831, 837
partial differential equations 820, 827f.
of polynomial deflation 363
in quadrature solution of Volterra equation 787f.
of recurrence relations 173ff., 177, 224f., 227f., 232, 247
and stiff differential equations 728f.
von Neumann analysis for PDEs 827f., 830, 833f., 840
see also Accuracy
Stabilized Kolmogorov-Smirnov test 621
Stabilizing functional 798
Staggered leapfrog method 833f.
Standard (probable) errors 1288, 1290
Standard deviation
of a distribution 605, 1269
of Fisher's z 632
of linear correlation coefficient 630
of sum squared difference of ranks 635, 1277
Standard (probable) errors 610, 656, 661, 667, 671, 684
Stars, as text separator 1009
Statement function, superseded by internal subprogram 1057, 1256
Statement labels 9
Statistical error 653
Statistical tests 603ff., 1269ff.
Anderson-Darling 621
average deviation 605, 1269
bootstrap method 686f.
chi-square 614f., 623ff., 1272, 1275f.

contingency coefficient C 625, 1275
contingency tables 622ff., 638, 1275f.
correlation 603f.
Cramer's V 625, 1275
difference of distributions 614ff., 1272
difference of means 609ff., 1269f.
difference of variances 611, 613, 1271
entropy measures of association 626ff., 1275f.
F-test 611, 613, 1271
Fisher's z-transformation 631f., 1276
general paradigm 603
Kendall's tau 634, 637ff., 1279
Kolmogorov-Smirnov 614, 617ff., 640, 694, 1273f., 1281
Kuiper's statistic 621
kurtosis 606, 608, 1269
L-estimates 694
linear correlation coefficient 630ff., 1276
M-estimates 694ff.
mean 603ff., 608ff., 1269f.
measures of association 604, 622ff., 1275
measures of central tendency 604ff., 1269
median 605, 694
mode 605
moments 604ff., 608, 1269
nonparametric correlation 633ff., 1277
Pearson's r 630ff., 1276
for periodic signal 570
phi statistic 625
R-estimates 694
rank correlation 633ff., 1277
robust 605, 634, 694ff.
semi-invariants 608
for shift vs. for spread 620f.
significance 609f., 1269ff.
significance, one- vs. two-sided 613, 632
skewness 606, 608, 1269
Spearman rank-order coefficient 634f., 694f., 1277
standard deviation 605, 1269
strength vs. significance 609f., 622
Student's t 610, 631, 1269
Student's t, for correlation 631
Student's t, paired samples 612, 1271
Student's t, Spearman rank-order coefficient 634, 1277
Student's t, unequal variances 611, 1270
sum squared difference of ranks 635, 1277
Tukey's trimean 694
two-dimensional 640, 1281ff.
variance 603ff., 607f., 612f., 1269ff.
Wilcoxon 694
see also Error; Robust estimation
Steak, without sizzle 809
Steed's method
Bessel functions 234, 239
continued fractions 164f.
Steepest descent method 414
in inverse problems 804
Step
doubling 130, 708f., 1052
tripling 136, 1055
Stieltjes, procedure of 151
Stiff equations 703, 727ff., 1308ff.
Kaps-Rentrop method 730, 1308
methods compared 739
predictor-corrector method 730
r.h.s. independent of x 729f.
Rosenbrock method 730, 1308
scaling of variables 730
semi-implicit extrapolation method 730, 1310f.
semi-implicit midpoint rule 735f., 1310f.
Stiff functions 100, 399
Stirling's approximation 206, 812
Stoermer's rule 726, 1307
Stopping criterion, in multigrid method 875f.
Stopping criterion, in polynomial root finding 366
Storage
band diagonal matrix 44, 1019
sparse matrices 71f., 1030
Storage association 2/xiv
Straight injection 867
Straight insertion 321f., 461f., 1167, 1227
Straight line fitting 655ff., 667f., 1285ff.
errors in both coordinates 660ff., 1286ff.
robust estimation 698, 1294ff.
Strassen's fast matrix algorithms 96f.
Stratified sampling, Monte Carlo 308f., 314
Stride (of an array) 944
communication bottleneck 969
Strongly implicit procedure (SIPSOL) 824
Structure constructor 2/xii
Structured programming 5ff.
Student's probability distribution 221f.
Student's t-test
for correlation 631
for difference of means 610, 1269
for difference of means (paired samples) 612, 1271
for difference of means (unequal variances) 611, 1270
for difference of ranks 635, 1277
Spearman rank-order coefficient 634, 1277
Sturmian sequence 469
Sub-random sequences *see* Quasi-random sequence
Subprogram 938
for data hiding 957, 1209, 1293, 1296
internal 954, 957, 1057, 1067, 1226, 1256
in module 940
undefined variables on exit 952f., 961, 1070, 1266, 1293, 1302
Subscript triplet (for array) 944
Subtraction, multiple precision 907, 1353
Subtractive method for random number generator 273, 1143
Subvector scaling 972, 974, 996, 1000
Successive over-relaxation (SOR) 857ff., 862, 1332f.
bad in multigrid method 866
Chebyshev acceleration 859f., 1332f.
choice of overrelaxation parameter 858
with logical mask 1333f.
parallelization 1333
sum() intrinsic function 945, 948, 966

Sum squared difference of ranks 634, 1277
Sums *see* Series
Sun 1/xxii, 2/xix, 886
  SPARCstation 1/xxii, 2/xix, 4
Supernova 1987A 640
SVD *see* Singular value decomposition (SVD)
swap() utility function 987, 990f., 1015, 1210
Symbol, of operator 866f.
Synthetic division 84, 167, 362, 370
  parallel algorithms 977ff., 999, 1048, 1071f., 1079, 1192
  repeated 978f.
Systematic errors 653

Tableau (interpolation) 103, 183
Tangent function, continued fraction 163
Target, for pointer 938f., 945, 952f.
Taylor series 180, 355f., 408, 702, 709, 742, 754, 759
Test programs 3
Thermodynamics, analogy for simulated annealing 437
Thinking Machines, Inc. 964
Threshold multiply of sparse matrices 74, 1031
Tides 560f.
Tikhonov-Miller regularization 799ff.
Time domain 490
Time splitting 847f., 861
tiny() intrinsic function 952
Toeplitz matrix 82, 85ff., 195, 1038
  LU decomposition 87
  new, fast algorithms 88f.
  nonsymmetric 86ff., 1038
Tongue twisters 333
Torus 297f., 304
Trade-off curve 795, 809
Trademarks 1/xxii, 2/xixf.
Transformation
  Gauss 256
  Landen 256
  method for random number generator 277ff.
Transformational functions 948ff.
Transforms, number theoretic 503f.
Transport error 831ff.
transpose() intrinsic function 950, 960, 969, 981, 1050, 1246
Transpose of sparse matrix 73f.
Trapezoidal rule 125, 127, 130ff., 134f., 579, 583, 782, 786, 1052, 1326f.
Traveling salesman problem 438ff., 1219ff.
Tridiagonal matrix 42, 63, 150, 453f., 488, 839f., 1018f.
  in alternating-direction implicit method (ADI) 861f.
  from cubic spline 109
  cyclic 67, 1030
  in cyclic reduction 853
  eigenvalues 469ff., 1228
  with fringes 822
  from operator splitting 861f.
  parallel algorithm 975, 1018, 1229f.
  recursive splitting 1229f.
  reduction of symmetric matrix to 462ff., 470, 1227f.
  serial algorithm 1018f.
  see also Matrix
Trigonometric
  functions, linear sequences 173
  functions, recurrence relation 172, 572
  functions, $\tan(\theta/2)$ as minimal 173
  interpolation 99
  solution of cubic equation 179f.
Truncation error 20f., 399, 709, 881, 1362
  in multigrid method 875
  in numerical derivatives 180
Tukey's biweight 697
Tukey's trimean 694
Turbo Pascal (Borland) 8
Twin errors 895
Two-dimensional *see* Multidimensional
Two-dimensional K–S test 640, 1281ff.
Two-pass algorithm for variance 607, 1269
Two-point boundary value problems 702, 745ff., 1314ff.
  automated allocation of mesh points 774f., 777
  boundary conditions 745ff., 749, 751f., 771, 1314ff.
  difficult cases 753, 1315f.
  eigenvalue problem for differential equations 748, 764ff., 770ff., 1319ff.
  free boundary problem 748, 776
  grid (mesh) points 746f., 754, 774f., 777
  internal boundary conditions 775ff.
  internal singular points 775ff.
  linear requires no iteration 751
  multiple shooting 753
  problems reducible to standard form 748
  regularity condition 775
  relaxation method 746f., 753ff., 1316ff.
  relaxation method, example of 764ff., 1319
  shooting to a fitting point 751ff., 1315f., 1323ff.
  shooting method 746, 749ff., 770ff., 1314ff., 1321ff.
  shooting method, example of 770ff., 1321ff.
  singular endpoints 751, 764, 771, 1315f., 1319ff.
  see also Elliptic partial differential equations
Two-sided exponential error distribution 696
Two-sided power spectral density 493
Two-step Lax-Wendroff method 835ff.
Two-volume edition, plan of 1/xiii
Two's complement arithmetic 1144
Type declarations, explicit vs. implicit 2

Ubound() intrinsic function 949
ULTRIX 1/xxiii, 2/xix
Uncertainty coefficient 628
Uncertainty principle 600
Undefined status, of arrays and pointers 952f., 961, 1070, 1266, 1293, 1302
Underflow, in IEEE arithmetic 883, 1343
Underrelaxation 857
Uniform deviates *see* Random deviates, uniform

Unitary (function) 843f.
Unitary (matrix) *see* Matrix
unit_matrix() utility function 985, 990, 1006, 1216, 1226, 1325
UNIX 1/xxiii, 2/viii, 2/xix, 4, 17, 276, 293, 886
Upper Hessenberg matrix *see* Hessenberg matrix
U.S. Postal Service barcode 894
unpack() intrinsic function 950, 964
  communication bottleneck 969
Upper subscript 944
upper_triangle() utility function 990, 1006, 1226, 1305
Upwind differencing 832f., 837
USE statement 936, 939f., 954, 957, 1067, 1384
USES keyword in program listings 2
Utility functions 987ff., 1364ff.
  add vector to matrix diagonal 1004, 1234, 1366, 1381
  alphabetical listing 988ff.
  argument checking 994f., 1370f.
  arithmetic progression 996, 1072, 1127, 1365, 1371f.
  array reallocation 992, 1070f., 1365, 1368f.
  assertion of numerical equality 995, 1022, 1365, 1370f.
  compared to intrinsics 990ff.
  complex *n*th root of unity 999f., 1379
  copying arrays 991, 1034, 1327f., 1365f.
  create unit matrix 1006, 1382
  cumulative product of an array 997f., 1072, 1086, 1375
  cumulative sum of an array 997, 1280f., 1365, 1375
  data types 1361
  elemental functions 1364
  error handling 994f., 1036, 1370f.
  generic functions 1364
  geometric progression 996f., 1365, 1372ff.
  get diagonal of matrix 1005, 1226f., 1366, 1381f.
  length of a vector 1008, 1383
  linear recurrence 996
  location in an array 992ff., 1015, 1017ff.
  location of first logical "true" 993, 1041, 1369
  location of maximum array value 993, 1015, 1017, 1365, 1369
  location of minimum array value 993, 1369f.
  logical assertion 994, 1086, 1090, 1092, 1365, 1370
  lower triangular mask 1007, 1200, 1382
  masked polynomial evaluation 1378
  masked swap of elements in two arrays 1368
  moving data 990ff., 1015
  multiply vector into matrix diagonal 1004f., 1366, 1381
  nrutil.f90 (module file) 1364ff.
  outer difference of vectors 1001, 1366, 1380
  outer logical and of vectors 1002
  outer operations on vectors 1000ff., 1379f.
  outer product of vectors 1000f., 1076, 1365f., 1379
  outer quotient of vectors 1001, 1379
  outer sum of vectors 1001, 1379f.
  overloading 1364
  partial cumulants of a polynomial 999, 1071, 1192f., 1365, 1378f.
  polynomial evaluation 996, 998f., 1258, 1365, 1376ff.
  scatter-with-add 1002f., 1032f., 1366, 1380f.
  scatter-with-combine 1002f., 1032f., 1380f.
  scatter-with-max 1003f., 1366, 1381
  set diagonal elements of matrix 1005, 1200, 1366, 1382
  skew operation on matrices 1004ff., 1381ff.
  swap elements of two arrays 991, 1015, 1365ff.
  upper triangular mask 1006, 1226, 1305, 1382

V-cycle 865, 1336
vabs() utility function 990, 1008, 1290
Validation of Numerical Recipes procedures 3f.
Valley, long or narrow 403, 407, 410
Van Cittert's method 804
Van Wijngaarden-Dekker-Brent method *see* Brent's method
Vandermonde matrix 82ff., 114, 1037, 1047
Variable length code 896, 1346ff.
Variable metric method 390, 418ff., 1215
  compared to conjugate gradient method 418
Variable step-size integration 123, 135, 703, 707ff., 720, 726, 731, 737, 742ff., 1298ff., 1303, 1308f., 1311ff.
Variance(s)
  correlation 605
  of distribution 603ff., 608, 611, 613, 1269
  pooled 610
  reduction of (in Monte Carlo) 299, 306ff.
  statistical differences between two 609, 1271
  two-pass algorithm for computing 607, 1269
  see also Covariance
Variational methods, partial differential equations 824
VAX 275, 293
Vector(s)
  length 1008, 1383
  norms 1036
  outer difference 1001, 1366, 1380
  outer operations 1000ff., 1379f.
  outer product 1000f., 1076, 1365f., 1379
Vector reduction 972, 977, 998
Vector subscripts 2/xiif., 984, 1002, 1032, 1034
  communication bottleneck 969, 981, 1250
VEGAS algorithm for Monte Carlo 309ff., 1161
Verhoeff's algorithm for checksums 894f., 1345

Viète's formulas for cubic roots 179
Vienna Fortran 2/xv
Virus, computer 889
Viscosity
  artificial 831, 837
  numerical 830f., 837
Visibility 956ff., 1209, 1293, 1296
VMS 1/xxii, 2/xix
Volterra equations 780f., 1326
  adaptive stepsize control 788
  analogy with ODEs 786
  block-by-block method 788
  first kind 781, 786
  nonlinear 781, 787
  second kind 781, 786ff., 1326f.
  unstable quadrature 787f.
von Neuman, John 963, 965
von Neumann-Richtmyer artificial viscosity 837
von Neumann stability analysis for PDEs 827f., 830, 833f., 840
Vowellish (coding example) 896f., 902

W-cycle 865, 1336
Warranty, disclaimer of 1/xx, 2/xvii
Wave equation 246, 818, 825f.
Wavelet transform 584ff., 1264ff.
  appearance of wavelets 590ff.
  approximation condition of order p 585
  coefficient values 586, 589, 1265
  contrasted with Fourier transform 584, 594
  Daubechies wavelet filter coefficients 584ff., 588, 590f., 594, 598, 1264ff.
  detail information 585
  discrete wavelet transform (DWT) 586f., 1264
  DWT (discrete wavelet transform) 586f., 1264ff.
  eliminating wrap-around 587
  fast solution of linear equations 597ff.
  filters 592f.
  and Fourier domain 592f.
  image processing 596f.
  for integral equations 782
  inverse 587
  Lemarie's wavelet 593
  of linear operator 597ff.
  mother-function coefficient 587
  mother functions 584
  multidimensional 595, 1267f.
  nonsmoothness of wavelets 591
  pyramidal algorithm 586, 1264
  quadrature mirror filter 585
  smooth information 585
  truncation 594f.
  wavelet filter coefficient 584, 587
  wavelets 584, 590ff.
Wavelets *see* Wavelet transform
Weber function 204
Weighted Kolmogorov-Smirnov test 621
Weighted least-squares fitting *see* Least squares fitting
Weighting, full vs. half in multigrid 867
Weights for Gaussian quadrature 140ff., 788f., 1059ff., 1328f.
  nonclassical weight function 151ff., 788f., 1064f., 1328f.
Welch window 547, 1254ff.
WG5 (ISO/IEC JTC1/SC22/WG5 Committee) 2/xiff.
where construct 943, 1291
  contrasted with merge 1023
  for iteration of a vector 1060
  nested 2/xv, 943, 960, 1100
  not MIMD 985
While iteration 13
Wiener filtering 535, 539ff., 558, 644
  compared to regularization 801
Wiener-Khinchin theorem 492, 558, 566f.
Wilcoxon test 694
Window function
  Bartlett 547, 1254ff.
  flat-topped 549
  Hamming 547
  Hann 547
  Parzen 547
  square 544, 546, 1254ff.
  Welch 547, 1254ff.
Windowing for spectral estimation 1255f.
Windows 95 2/xix
Windows NT 2/xix
Winograd Fourier transform algorithms 503
Woodbury formula 68ff., 83
Wordlength 18
Workspace, reallocation in Fortran 90 1070f.
World Wide Web, Numerical Recipes site 1/xx, 2/xvii
Wraparound
  in integer arithmetic 1146, 1148
  order for storing spectrum 501
  problem in convolution 533
Wronskian, of Bessel functions 234, 239

X.25 protocol 890
X3J3 Committee 2/viii, 2/xff., 2/xv, 947, 959, 964, 968, 990
XMODEM checksum 889
X-ray diffraction pattern, processing of 805

Yale Sparse Matrix Package 64, 71

Z-transform 554, 559, 565
Z-transformation, Fisher's 631f., 1276
Zaman, A. 1149
Zealots 814
Zebra relaxation 866
Zero contours 372
Zero-length array 944
Zeroth-order regularization 796ff.
Zip code, barcode for 894
Ziv-Lempel compression 896
zroots_unity() utility function 974, 990, 999