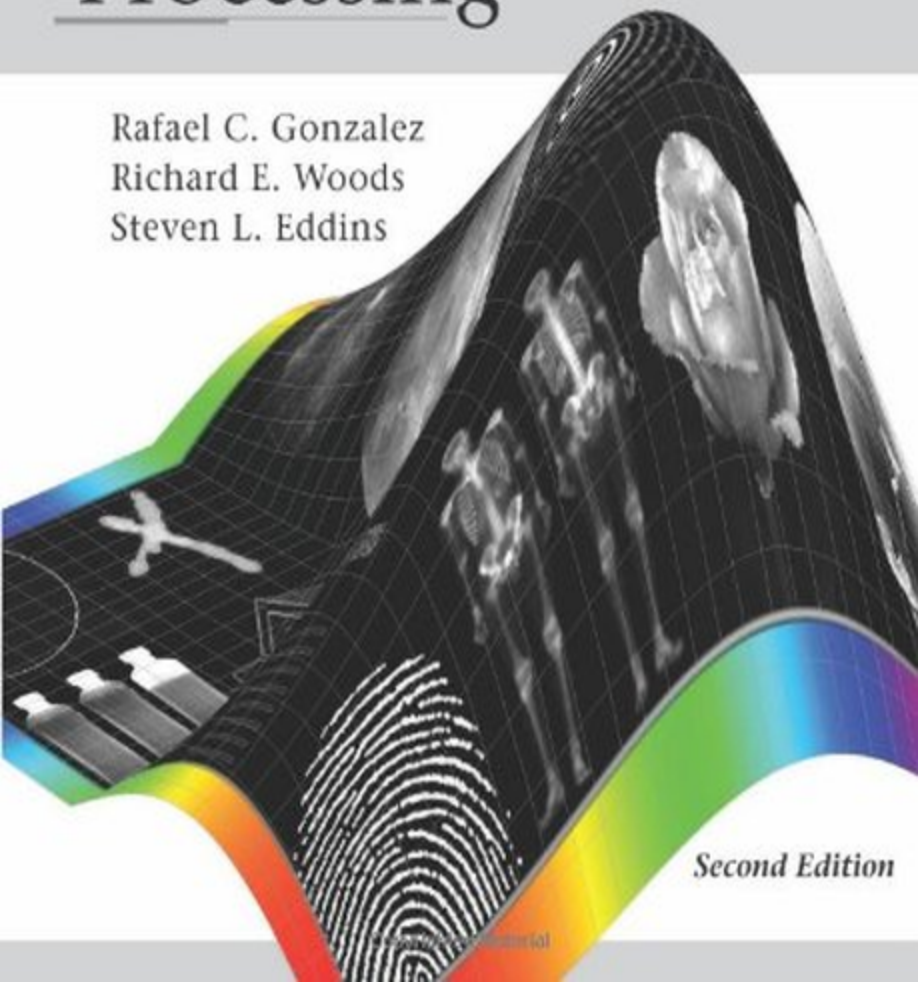
Copyrighted Material

Digital Image Processing USING MATLAB®

Rafael C. Gonzalez
Richard E. Woods
Steven L. Eddins



Second Edition

Copyrighted Material



Digital Image Processing Using MATLAB®

Second Edition

Rafael C. Gonzalez
University of Tennessee

Richard E. Woods
MedData Interactive

Steven L. Eddins
The MathWorks, Inc.

2009

Gatesmark Publishing®
A Division of Gatesmark,® LLC
www.gatesmark.com

Library of Congress Cataloging-in-Publication Data on File

Library of Congress Control Number: 2009902793

TA1632
G66
2009



Gatesmark Publishing
A Division of Gatesmark, LLC
www.gatesmark.com

© 2009 by Gatesmark, LLC

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, without written permission from the publisher.

Gatesmark Publishing® is a registered trademark of Gatesmark, LLC, www.gatesmark.com.

Gatesmark® is a registered trademark of Gatesmark, LLC, www.gatesmark.com.

MATLAB® is a registered trademark of The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher shall not be liable in any event for incidental or consequential damages with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 978-0-9820854-0-0

To Ryan
To Janice, David, and Jonathan
and
To Geri, Christopher, and Nicholas

Contents

Preface xi
Acknowledgements xiii
About the Authors xv

1 *Introduction 1*

Preview 1
1.1 Background 1
1.2 What Is Digital Image Processing? 2
1.3 Background on MATLAB and the Image Processing Toolbox 4
1.4 Areas of Image Processing Covered in the Book 5
1.5 The Book Web Site 7
1.6 Notation 7
1.7 The MATLAB Desktop 7
1.7.1 Using the MATLAB Editor/Debugger 10
1.7.2 Getting Help 10
1.7.3 Saving and Retrieving Work Session Data 11
1.8 How References Are Organized in the Book 11
Summary 12

2 *Fundamentals 13*

Preview 13
2.1 Digital Image Representation 13
2.1.1 Coordinate Conventions 14
2.1.2 Images as Matrices 15
2.2 Reading Images 15
2.3 Displaying Images 18
2.4 Writing Images 21
2.5 Classes 26
2.6 Image Types 27
2.6.1 Gray-scale Images 27
2.6.2 Binary Images 27
2.6.3 A Note on Terminology 28
2.7 Converting between Classes 28
2.8 Array Indexing 33
2.8.1 Indexing Vectors 33
2.8.2 Indexing Matrices 35
2.8.3 Indexing with a Single Colon 37
2.8.4 Logical Indexing 38
2.8.5 Linear Indexing 39
2.8.6 Selecting Array Dimensions 42

2.8.7 Sparse Matrices 42
2.9 Some Important Standard Arrays 43
2.10 Introduction to M-Function Programming 44
2.10.1 M-Files 44
2.10.2 Operators 46
2.10.3 Flow Control 57
2.10.4 Function Handles 63
2.10.5 Code Optimization 65
2.10.6 Interactive I/O 71
2.10.7 An Introduction to Cell Arrays and Structures 74
Summary 79

3 *Intensity Transformations and Spatial Filtering 80*
Preview 80
3.1 Background 80
3.2 Intensity Transformation Functions 81
3.2.1 Functions `imadjust` and `stretchlim` 82
3.2.2 Logarithmic and Contrast-Stretching Transformations 84
3.2.3 Specifying Arbitrary Intensity Transformations 86
3.2.4 Some Utility M-functions for Intensity Transformations 87
3.3 Histogram Processing and Function Plotting 93
3.3.1 Generating and Plotting Image Histograms 94
3.3.2 Histogram Equalization 99
3.3.3 Histogram Matching (Specification) 102
3.3.4 Function `adapthisteq` 107
3.4 Spatial Filtering 109
3.4.1 Linear Spatial Filtering 109
3.4.2 Nonlinear Spatial Filtering 117
3.5 Image Processing Toolbox Standard Spatial Filters 120
3.5.1 Linear Spatial Filters 120
3.5.2 Nonlinear Spatial Filters 124
3.6 Using Fuzzy Techniques for Intensity Transformations and Spatial Filtering 128
3.6.1 Background 128
3.6.2 Introduction to Fuzzy Sets 128
3.6.3 Using Fuzzy Sets 133
3.6.4 A Set of Custom Fuzzy M-functions 140
3.6.5 Using Fuzzy Sets for Intensity Transformations 155
3.6.6 Using Fuzzy Sets for Spatial Filtering 158
Summary 163

4 *Filtering in the Frequency Domain 164*
Preview 164

4.1 The 2-D Discrete Fourier Transform 164
4.2 Computing and Visualizing the 2-D DFT in MATLAB 168
4.3 Filtering in the Frequency Domain 172
4.3.1 Fundamentals 173
4.3.2 Basic Steps in DFT Filtering 178
4.3.3 An M-function for Filtering in the Frequency Domain 179
4.4 Obtaining Frequency Domain Filters from Spatial Filters 180
4.5 Generating Filters Directly in the Frequency Domain 185
4.5.1 Creating Meshgrid Arrays for Use in Implementing Filters in the Frequency Domain 186
4.5.2 Lowpass (Smoothing) Frequency Domain Filters 187
4.5.3 Wireframe and Surface Plotting 190
4.6 Highpass (Sharpening) Frequency Domain Filters 194
4.6.1 A Function for Highpass Filtering 194
4.6.2 High-Frequency Emphasis Filtering 197
4.7 Selective Filtering 199
4.7.1 Bandreject and Bandpass Filters 199
4.7.2 Notchreject and Notchpass Filters 202
Summary 208

5 *Image Restoration and Reconstruction 209*
Preview 209
5.1 A Model of the Image Degradation/Restoration Process 210
5.2 Noise Models 211
5.2.1 Adding Noise to Images with Function `imnoise` 211
5.2.2 Generating Spatial Random Noise with a Specified Distribution 212
5.2.3 Periodic Noise 220
5.2.4 Estimating Noise Parameters 224
5.3 Restoration in the Presence of Noise Only—Spatial Filtering 229
5.3.1 Spatial Noise Filters 229
5.3.2 Adaptive Spatial Filters 233
5.4 Periodic Noise Reduction Using Frequency Domain Filtering 236
5.5 Modeling the Degradation Function 237
5.6 Direct Inverse Filtering 240
5.7 Wiener Filtering 240
5.8 Constrained Least Squares (Regularized) Filtering 244
5.9 Iterative Nonlinear Restoration Using the Lucy-Richardson Algorithm 246
5.10 Blind Deconvolution 250
5.11 Image Reconstruction from Projections 251
5.11.1 Background 252
5.11.2 Parallel-Beam Projections and the Radon Transform 254
5.11.3 The Fourier Slice Theorem and Filtered Backprojections 257
5.11.4 Filter Implementation 258

5.11.5 Reconstruction Using Fan-Beam Filtered Backprojections 259
5.11.6 Function `radon` 260
5.11.7 Function `iradon` 263
5.11.8 Working with Fan-Beam Data 268
Summary 277

6 *Geometric Transformations and Image Registration* 278
Preview 278
6.1 Transforming Points 278
6.2 Affine Transformations 283
6.3 Projective Transformations 287
6.4 Applying Geometric Transformations to Images 288
6.5 Image Coordinate Systems in MATLAB 291
6.5.1 Output Image Location 293
6.5.2 Controlling the Output Grid 297
6.6 Image Interpolation 299
6.6.1 Interpolation in Two Dimensions 302
6.6.2 Comparing Interpolation Methods 302
6.7 Image Registration 305
6.7.1 Registration Process 306
6.7.2 Manual Feature Selection and Matching Using `cpselect` 306
6.7.3 Inferring Transformation Parameters Using `cp2tform` 307
6.7.4 Visualizing Aligned Images 307
6.7.5 Area-Based Registration 311
6.7.5 Automatic Feature-Based Registration 316
Summary 317

7 *Color Image Processing* 318
Preview 318
7.1 Color Image Representation in MATLAB 318
7.1.1 RGB Images 318
7.1.2 Indexed Images 321
7.1.3 Functions for Manipulating RGB and Indexed Images 323
7.2 Converting Between Color Spaces 328
7.2.1 NTSC Color Space 328
7.2.2 The YCbCr Color Space 329
7.2.3 The HSV Color Space 329
7.2.4 The CMY and CMYK Color Spaces 330
7.2.5 The HSI Color Space 331
7.2.6 Device-Independent Color Spaces 340
7.3 The Basics of Color Image Processing 349
7.4 Color Transformations 350
7.5 Spatial Filtering of Color Images 360

7.5.1 Color Image Smoothing 360
7.5.2 Color Image Sharpening 365
7.6 Working Directly in RGB Vector Space 366
7.6.1 Color Edge Detection Using the Gradient 366
7.6.2 Image Segmentation in RGB Vector Space 372
Summary 376

8 *Wavelets 377*
Preview 377
8.1 Background 377
8.2 The Fast Wavelet Transform 380
8.2.1 FWTs Using the Wavelet Toolbox 381
8.2.2 FWTs without the Wavelet Toolbox 387
8.3 Working with Wavelet Decomposition Structures 396
8.3.1 Editing Wavelet Decomposition Coefficients without the Wavelet Toolbox 399
8.3.2 Displaying Wavelet Decomposition Coefficients 404
8.4 The Inverse Fast Wavelet Transform 408
8.5 Wavelets in Image Processing 414
Summary 419

9 *Image Compression 420*
Preview 420
9.1 Background 421
9.2 Coding Redundancy 424
9.2.1 Huffman Codes 427
9.2.2 Huffman Encoding 433
9.2.3 Huffman Decoding 439
9.3 Spatial Redundancy 446
9.4 Irrelevant Information 453
9.5 JPEG Compression 456
9.5.1 JPEG 456
9.5.2 JPEG 2000 464
9.6 Video Compression 472
9.6.1 MATLAB Image Sequences and Movies 473
9.6.2 Temporal Redundancy and Motion Compensation 476
Summary 485

10 *Morphological Image Processing 486*
Preview 486
10.1 Preliminaries 487
10.1.1 Some Basic Concepts from Set Theory 487
10.1.2 Binary Images, Sets, and Logical Operators 489
10.2 Dilation and Erosion 490

10.2.1 Dilation 490
10.2.2 Structuring Element Decomposition 493
10.2.3 The `strel` Function 494
10.2.4 Erosion 497
10.3 Combining Dilation and Erosion 500
10.3.1 Opening and Closing 500
10.3.2 The Hit-or-Miss Transformation 503
10.3.3 Using Lookup Tables 506
10.3.4 Function `bwmorph` 511
10.4 Labeling Connected Components 514
10.5 Morphological Reconstruction 518
10.5.1 Opening by Reconstruction 518
10.5.2 Filling Holes 520
10.5.3 Clearing Border Objects 521
10.6 Gray-Scale Morphology 521
10.6.1 Dilation and Erosion 521
10.6.2 Opening and Closing 524
10.6.3 Reconstruction 530
Summary 534

11 *Image Segmentation 535*
Preview 535
11.1 Point, Line, and Edge Detection 536
11.1.1 Point Detection 536
11.1.2 Line Detection 538
11.1.3 Edge Detection Using Function `edge` 541
11.2 Line Detection Using the Hough Transform 549
11.2.1 Background 551
11.2.2 Toolbox Hough Functions 552
11.3 Thresholding 557
11.3.1 Foundation 557
11.3.2 Basic Global Thresholding 559
11.3.3 Optimum Global Thresholding Using Otsu's Method 561
11.3.4 Using Image Smoothing to Improve Global Thresholding 565
11.3.5 Using Edges to Improve Global Thresholding 567
11.3.6 Variable Thresholding Based on Local Statistics 571
11.3.7 Image Thresholding Using Moving Averages 575
11.4 Region-Based Segmentation 578
11.4.1 Basic Formulation 578
11.4.2 Region Growing 578
11.4.3 Region Splitting and Merging 582
11.5 Segmentation Using the Watershed Transform 588
11.5.1 Watershed Segmentation Using the Distance Transform 589
11.5.2 Watershed Segmentation Using Gradients 591
11.5.3 Marker-Controlled Watershed Segmentation 593

Summary 596

12 *Representation and Description* 597

Preview 597

12.1 Background 597

12.1.1 Functions for Extracting Regions and Their Boundaries 598

12.1.2 Some Additional MATLAB and Toolbox Functions Used in This Chapter 603

12.1.3 Some Basic Utility M-Functions 604

12.2 Representation 606

12.2.1 Chain Codes 606

12.2.2 Polygonal Approximations Using Minimum-Perimeter Polygons 610

12.2.3 Signatures 619

12.2.4 Boundary Segments 622

12.2.5 Skeletons 623

12.3 Boundary Descriptors 625

12.3.1 Some Simple Descriptors 625

12.3.2 Shape Numbers 626

12.3.3 Fourier Descriptors 627

12.3.4 Statistical Moments 632

12.3.5 Corners 633

12.4 Regional Descriptors 641

12.4.1 Function `regionprops` 642

12.4.2 Texture 644

12.4.3 Moment Invariants 656

12.5 Using Principal Components for Description 661

Summary 672

13 *Object Recognition* 674

Preview 674

13.1 Background 674

13.2 Computing Distance Measures in MATLAB 675

13.3 Recognition Based on Decision-Theoretic Methods 679

13.3.1 Forming Pattern Vectors 680

13.3.2 Pattern Matching Using Minimum-Distance Classifiers 680

13.3.3 Matching by Correlation 681

13.3.4 Optimum Statistical Classifiers 684

13.3.5 Adaptive Learning Systems 691

13.4 Structural Recognition 691

13.4.1 Working with Strings in MATLAB 692

13.4.2 String Matching 701

Summary 706

Appendix A *M-Function Summary* 707

Appendix B *ICE and MATLAB Graphical User Interfaces* 724

Appendix C *Additional Custom M-functions* 750

Bibliography 813

Index 817

Preface

This edition of *Digital Image Processing Using MATLAB* is a major revision of the book. As in the previous edition, the focus of the book is based on the fact that solutions to problems in the field of digital image processing generally require extensive experimental work involving software simulation and testing with large sets of sample images. Although algorithm development typically is based on theoretical underpinnings, the actual implementation of these algorithms almost always requires parameter estimation and, frequently, algorithm revision and comparison of candidate solutions. Thus, selection of a flexible, comprehensive, and well-documented software development environment is a key factor that has important implications in the cost, development time, and portability of image processing solutions.

Despite its importance, surprisingly little has been written on this aspect of the field in the form of textbook material dealing with both theoretical principles and software implementation of digital image processing concepts. The first edition of this book was written in 2004 to meet just this need. This new edition of the book continues the same focus. Its main objective is to provide a foundation for implementing image processing algorithms using modern software tools. A complementary objective is that the book be self-contained and easily readable by individuals with a basic background in digital image processing, mathematical analysis, and computer programming, all at a level typical of that found in a junior/senior curriculum in a technical discipline. Rudimentary knowledge of MATLAB also is desirable.

To achieve these objectives, we felt that two key ingredients were needed. The first was to select image processing material that is representative of material covered in a formal course of instruction in this field. The second was to select software tools that are well supported and documented, and which have a wide range of applications in the "real" world.

To meet the first objective, most of the theoretical concepts in the following chapters were selected from *Digital Image Processing* by Gonzalez and Woods, which has been the choice introductory textbook used by educators all over the world for over three decades. The software tools selected are from the MATLAB® Image Processing Toolbox™, which similarly occupies a position of eminence in both education and industrial applications. A basic strategy followed in the preparation of the current edition was to continue providing a seamless integration of well-established theoretical concepts and their implementation using state-of-the-art software tools.

The book is organized along the same lines as *Digital Image Processing*. In this way, the reader has easy access to a more detailed treatment of all the image processing concepts discussed here, as well as an up-to-date set of references for further reading. Following this approach made it possible to present theoretical material in a succinct manner and thus we were able to maintain a focus on the software implementation aspects of image processing problem solutions. Because it works in the MATLAB computing environment, the Image Processing Toolbox offers some significant advantages, not only in the breadth of its computational

tools, but also because it is supported under most operating systems in use today. A unique feature of this book is its emphasis on showing how to develop new code to enhance existing MATLAB and toolbox functionality. This is an important feature in an area such as image processing, which, as noted earlier, is characterized by the need for extensive algorithm development and experimental work.

After an introduction to the fundamentals of MATLAB functions and programming, the book proceeds to address the mainstream areas of image processing. The major areas covered include intensity transformations, fuzzy image processing, linear and nonlinear spatial filtering, the frequency domain filtering, image restoration and reconstruction, geometric transformations and image registration, color image processing, wavelets, image data compression, morphological image processing, image segmentation, region and boundary representation and description, and object recognition. This material is complemented by numerous illustrations of how to solve image processing problems using MATLAB and toolbox functions. In cases where a function did not exist, a new function was written and documented as part of the instructional focus of the book. Over 120 new functions are included in the following chapters. These functions increase the scope of the Image Processing Toolbox by approximately 40% and also serve the important purpose of further illustrating how to implement new image processing software solutions.

The material is presented in textbook format, not as a software manual. Although the book is self-contained, we have established a companion web site (see Section 1.5) designed to provide support in a number of areas. For students following a formal course of study or individuals embarked on a program of self study, the site contains tutorials and reviews on background material, as well as projects and image databases, including all images in the book. For instructors, the site contains classroom presentation materials that include PowerPoint slides of all the images and graphics used in the book. Individuals already familiar with image processing and toolbox fundamentals will find the site a useful place for up-to-date references, new implementation techniques, and a host of other support material not easily found elsewhere. All purchasers of new books are eligible to download executable files of all the new functions developed in the text at no cost.

As is true of most writing efforts of this nature, progress continues after work on the manuscript stops. For this reason, we devoted significant effort to the selection of material that we believe is fundamental, and whose value is likely to remain applicable in a rapidly evolving body of knowledge. We trust that readers of the book will benefit from this effort and thus find the material timely and useful in their work.

Rafael C. Gonzalez
Richard E. Woods
Steven L. Eddins

Acknowledgements

We are indebted to a number of individuals in academic circles as well as in industry and government who have contributed to the preparation of the book. Their contributions have been important in so many different ways that we find it difficult to acknowledge them in any other way but alphabetically. We wish to extend our appreciation to Mongi A. Abidi, Peter J. Acklam, Serge Beucher, Ernesto Bribiesca, Michael W. Davidson, Courtney Esposito, Naomi Fernandes, Susan L. Forsburg, Thomas R. Gest, Chris Griffin, Daniel A. Hammer, Roger Heady, Brian Johnson, Mike Karr, Lisa Kempler, Roy Lurie, Jeff Mather, Eugene McGoldrick, Ashley Mohamed, Joseph E. Pascente, David R. Pickens, Edgardo Felipe Riveron, Michael Robinson, Brett Shoelson, Loren Shure, Inpakala Simon, Jack Sklanski, Sally Stowe, Craig Watson, Greg Wolodkin, and Mara Yale. We also wish to acknowledge the organizations cited in the captions of many of the figures in the book for their permission to use that material.

R.C.G
R.E.W
S.L.E

The Book Web Site

Digital Image Processing Using MATLAB is a self-contained book. However, the companion web site at

www.ImageProcessingPlace.com

offers additional support in a number of important areas.

For the Student or Independent Reader the site contains

- Reviews in areas such as MATLAB, probability, statistics, vectors, and matrices.
- Sample computer projects.
- A Tutorials section containing dozens of tutorials on most of the topics discussed in the book.
- A database containing all the images in the book.

For the Instructor the site contains

- Classroom presentation materials in PowerPoint format.
- Numerous links to other educational resources.

For the Practitioner the site contains additional specialized topics such as

- Links to commercial sites.
- Selected new references.
- Links to commercial image databases.

The web site is an ideal tool for keeping the book current between editions by including new topics, digital images, and other relevant material that has appeared after the book was published. Although considerable care was taken in the production of the book, the web site is also a convenient repository for any errors that may be discovered between printings.

About the Authors

Rafael C. Gonzalez

R. C. Gonzalez received the B.S.E.E. degree from the University of Miami in 1965 and the M.E. and Ph.D. degrees in electrical engineering from the University of Florida, Gainesville, in 1967 and 1970, respectively. He joined the Electrical Engineering and Computer Science Department at the University of Tennessee, Knoxville (UTK) in 1970, where he became Associate Professor in 1973, Professor in 1978, and Distinguished Service Professor in 1984. He served as Chairman of the department from 1994 through 1997. He is currently a Professor Emeritus of Electrical and Computer Science at UTK.

He is the founder of the Image & Pattern Analysis Laboratory and the Robotics & Computer Vision Laboratory at the University of Tennessee. He also founded Perceptics Corporation in 1982 and was its president until 1992. The last three years of this period were spent under a full-time employment contract with Westinghouse Corporation, who acquired the company in 1989. Under his direction, Perceptics became highly successful in image processing, computer vision, and laser disk storage technologies. In its initial ten years, Perceptics introduced a series of innovative products, including: The world's first commercially-available computer vision system for automatically reading the license plate on moving vehicles; a series of large-scale image processing and archiving systems used by the U.S. Navy at six different manufacturing sites throughout the country to inspect the rocket motors of missiles in the Trident II Submarine Program; the market leading family of imaging boards for advanced Macintosh computers; and a line of trillion-byte laser disk products.

He is a frequent consultant to industry and government in the areas of pattern recognition, image processing, and machine learning. His academic honors for work in these fields include the 1977 UTK College of Engineering Faculty Achievement Award; the 1978 UTK Chancellor's Research Scholar Award; the 1980 Magnavox Engineering Professor Award; and the 1980 M. E. Brooks Distinguished Professor Award. In 1981 he became an IBM Professor at the University of Tennessee and in 1984 he was named a Distinguished Service Professor there. He was awarded a Distinguished Alumnus Award by the University of Miami in 1985, the Phi Kappa Phi Scholar Award in 1986, and the University of Tennessee's Nathan W. Dougherty Award for Excellence in Engineering in 1992. Honors for industrial accomplishment include the 1987 IEEE Outstanding Engineer Award for Commercial Development in Tennessee; the 1988 Albert Rose National Award for Excellence in Commercial Image Processing; the 1989 B. Otto Wheeley Award for Excellence in Technology Transfer; the 1989 Coopers and Lybrand Entrepreneur of the Year Award; the 1992 IEEE Region 3 Outstanding Engineer Award; and the 1993 Automated Imaging Association National Award for Technology Development.

Dr. Gonzalez is author or coauthor of over 100 technical articles, two edited books, and five textbooks in the fields of pattern recognition, image processing, and robotics. His books are used in over 1000 universities and research institutions throughout the world. He is listed in the prestigious Marquis *Who's Who in America*, Marquis *Who's Who in Engineering*, Marquis *Who's Who in the World*, and in 10

other national and international biographical citations. He is the co-holder of two U.S. Patents, and has been an associate editor of the *IEEE Transactions on Systems, Man and Cybernetics*, and the *International Journal of Computer and Information Sciences*. He is a member of numerous professional and honorary societies, including Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Sigma Xi. He is a Fellow of the IEEE.

Richard E. Woods

Richard E. Woods earned his B.S., M.S., and Ph.D. degrees in Electrical Engineering from the University of Tennessee, Knoxville. His professional experiences range from entrepreneurial to the more traditional academic, consulting, governmental, and industrial pursuits. Most recently, he founded MedData Interactive, a high technology company specializing in the development of handheld computer systems for medical applications. He was also a founder and Vice President of Perceptics Corporation, where he was responsible for the development of many of the company's quantitative image analysis and autonomous decision making products.

Prior to Perceptics and MedData, Dr. Woods was an Assistant Professor of Electrical Engineering and Computer Science at the University of Tennessee and prior to that, a computer applications engineer at Union Carbide Corporation. As a consultant, he has been involved in the development of a number of special-purpose digital processors for a variety of space and military agencies, including NASA, the Ballistic Missile Systems Command, and the Oak Ridge National Laboratory.

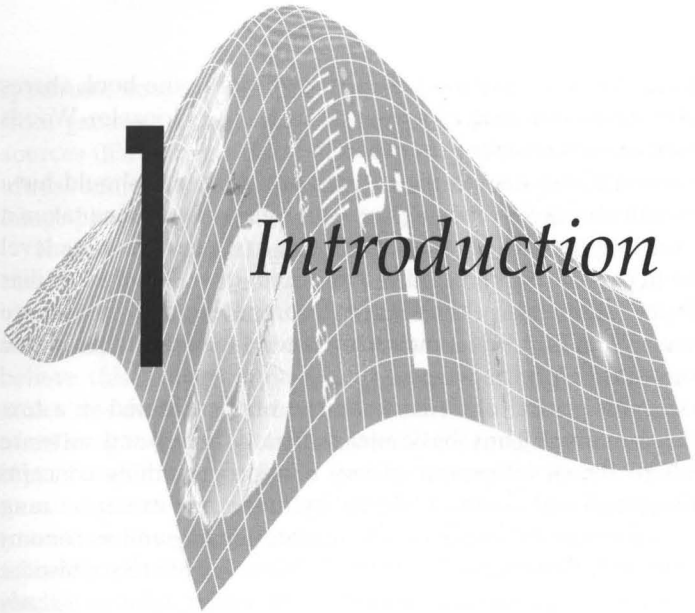
Dr. Woods has published numerous articles related to digital signal processing and is coauthor of *Digital Image Processing*, the leading text in the field. He is a member of several professional societies, including Tau Beta Pi, Phi Kappa Phi, and the IEEE. In 1986, he was recognized as a Distinguished Engineering Alumnus of the University of Tennessee.

Steven L. Eddins

Steven L. Eddins is development manager of the image processing group at The MathWorks, Inc. He led the development of several versions of the company's Image Processing Toolbox. His professional interests include building software tools that are based on the latest research in image processing algorithms, and that have a broad range of scientific and engineering applications.

Prior to joining The MathWorks, Inc. in 1993, Dr. Eddins was on the faculty of the Electrical Engineering and Computer Science Department at the University of Illinois, Chicago. There he taught graduate and senior-level classes in digital image processing, computer vision, pattern recognition, and filter design, and he performed research in the area of image compression.

Dr. Eddins holds a B.E.E. (1986) and a Ph.D. (1990), both in electrical engineering from the Georgia Institute of Technology. He is a senior member of the IEEE.



1 Introduction

Preview

Digital image processing is an area characterized by the need for extensive experimental work to establish the viability of proposed solutions to a given problem. In this chapter, we outline how a theoretical foundation and state-of-the-art software can be integrated into a prototyping environment whose objective is to provide a set of well-supported tools for the solution of a broad class of problems in digital image processing.

1.1 Background

An important characteristic underlying the design of image processing systems is the significant level of testing and experimentation that normally is required before arriving at an acceptable solution. This characteristic implies that the ability to formulate approaches and quickly prototype candidate solutions generally plays a major role in reducing the cost and time required to arrive at a viable system implementation.

Little has been written in the way of instructional material to bridge the gap between theory and application in a well-supported software environment for image processing. The main objective of this book is to integrate under one cover a broad base of theoretical concepts with the knowledge required to implement those concepts using state-of-the-art image processing software tools. The theoretical underpinnings of the material in the following chapters are based on the leading textbook in the field: *Digital Image Processing*, by Gonzalez and Woods.[†] The software code and supporting tools are based on the leading software in the field: *MATLAB*® and the *Image Processing Toolbox*™

[†] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 2008 .

from The MathWorks, Inc. (see Section 1.3). The material in the book shares the same design, notation, and style of presentation as the Gonzalez-Woods text, thus simplifying cross-referencing between the two.

The book is self-contained. To master its contents, a reader should have introductory preparation in digital image processing, either by having taken a formal course of study on the subject at the senior or first-year graduate level, or by acquiring the necessary background in a program of self-study. Familiarity with MATLAB and rudimentary knowledge of computer programming are assumed also. Because MATLAB is a matrix-oriented language, basic knowledge of matrix analysis is helpful.

The book is based on principles. It is organized and presented in a textbook format, not as a manual. Thus, basic ideas of both theory and software are explained prior to the development of any new programming concepts. The material is illustrated and clarified further by numerous examples ranging from medicine and industrial inspection to remote sensing and astronomy. This approach allows orderly progression from simple concepts to sophisticated implementation of image processing algorithms. However, readers already familiar with MATLAB, the Image Processing Toolbox, and image processing fundamentals can proceed directly to specific applications of interest, in which case the functions in the book can be used as an extension of the family of toolbox functions. All new functions developed in the book are fully documented, and the code for each is included either in a chapter or in Appendix C.

We use the term *custom function* to denote a function developed in the book, as opposed to a "standard" MATLAB or Image Processing Toolbox function.

Over 120 *custom functions* are developed in the chapters that follow. These functions extend by nearly 45% the set of about 270 functions in the Image Processing Toolbox. In addition to addressing specific applications, the new functions are good examples of how to combine existing MATLAB and toolbox functions with new code to develop prototype solutions to a broad spectrum of problems in digital image processing. The toolbox functions, as well as the functions developed in the book, run under most operating systems. Consult the book web site (see Section 1.5) for a complete list.

1.2 What Is Digital Image Processing?

An image may be defined as a two-dimensional function, $f(x, y)$, where x and y are *spatial coordinates*, and the amplitude of f at any pair of coordinates (x, y) is called the *intensity* or *gray level* of the image at that point. When x, y, and the amplitude values of f are all finite, discrete quantities, we call the image a *digital image*. The field of digital image processing refers to processing digital images by means of a digital computer. Note that a digital image is composed of a finite number of elements, each of which has a particular location and value. These elements are referred to as *picture elements*, *image elements*, *pels*, and *pixels*. *Pixel* is the term used most widely to denote the elements of a digital image. We consider these definitions formally in Chapter 2.

Vision is the most advanced of our senses, so it is not surprising that images play the single most important role in human perception. However, unlike humans, who are limited to the visual band of the electromagnetic (EM)

spectrum, imaging machines cover almost the entire EM spectrum, ranging from gamma to radio waves. They can operate also on images generated by sources that humans do not customarily associate with images. These include ultrasound, electron microscopy, and computer-generated images. Thus, digital image processing encompasses a wide and varied field of applications.

There is no general agreement among authors regarding where image processing stops and other related areas, such as image analysis and computer vision, begin. Sometimes a distinction is made by defining image processing as a discipline in which both the input and output of a process are images. We believe this to be a limiting and somewhat artificial boundary. For example, under this definition, even the trivial task of computing the average intensity of an image would not be considered an image processing operation. On the other hand, there are fields, such as computer vision, whose ultimate goal is to use computers to emulate human vision, including learning and being able to make inferences and take actions based on visual inputs. This area itself is a branch of artificial intelligence (AI), whose objective is to emulate human intelligence. The field of AI is in its infancy in terms of practical developments, with progress having been much slower than originally anticipated. The area of *image analysis* (also called *image understanding*) is in between image processing and computer vision.

There are no clear-cut boundaries in the continuum from image processing at one end to computer vision at the other. However, a useful paradigm is to consider three types of computerized processes in this continuum: low-, mid-, and high-level processes. *Low-level* processes involve primitive operations, such as image preprocessing to reduce noise, contrast enhancement, and image sharpening. A low-level process is characterized by the fact that both its inputs and outputs typically are images. *Mid-level* processes on images involve tasks such as segmentation (partitioning an image into regions or objects), description of those objects to reduce them to a form suitable for computer processing, and classification (recognition) of individual objects. A mid-level process is characterized by the fact that its inputs generally are images, but its outputs are attributes extracted from those images (e.g., edges, contours, and the identity of individual objects). Finally, *high-level* processing involves "making sense" of an ensemble of recognized objects, as in image analysis, and, at the far end of the continuum, performing the cognitive functions normally associated with human vision.

Based on the preceding comments, we see that a logical place of overlap between image processing and image analysis is the area of recognition of individual regions or objects in an image. Thus, what we call in this book *digital image processing* encompasses processes whose inputs and outputs are images and, in addition, encompasses processes that extract attributes from images, up to and including the recognition of individual objects. As a simple illustration to clarify these concepts, consider the area of automated analysis of text. The processes of acquiring an image of a region containing the text, preprocessing that image, extracting (segmenting) the individual characters, describing the characters in a form suitable for computer processing, and recognizing those

individual characters, are in the scope of what we call digital image processing in this book. Making sense of the content of the page may be viewed as being in the domain of image analysis and even computer vision, depending on the level of complexity implied by the statement "making sense of." Digital image processing, as we have defined it, is used successfully in a broad range of areas of exceptional social and economic value.

1.3 Background on MATLAB and the Image Processing Toolbox

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include the following:

- Math and computation
- Algorithm development
- Data acquisition
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including building graphical user interfaces

MATLAB is an interactive system whose basic data element is a matrix. This allows formulating solutions to many technical computing problems, especially those involving matrix representations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C.

The name MATLAB stands for *Matrix Laboratory*. MATLAB was written originally to provide easy access to matrix and linear algebra software that previously required writing FORTRAN programs to use. Today, MATLAB incorporates state of the art numerical computation software that is highly optimized for modern processors and memory architectures.

In university environments, MATLAB is the standard computational tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the computational tool of choice for research, development, and analysis. MATLAB is complemented by a family of application-specific solutions called *toolboxes*. The Image Processing Toolbox is a collection of MATLAB functions (called *M-functions* or *M-files*) that extend the capability of the MATLAB environment for the solution of digital image processing problems. Other toolboxes that sometimes are used to complement the Image Processing Toolbox are the Signal Processing, Neural Networks, Fuzzy Logic, and Wavelet Toolboxes.

As we discuss in more detail in Chapter 2, images may be treated as matrices, thus making MATLAB software a natural choice for image processing applications.

The *MATLAB & Simulink Student Version* is a product that includes a full-featured version of MATLAB, the Image Processing Toolbox, and several other useful toolboxes. The Student Version can be purchased at significant discounts at university bookstores and at the MathWorks web site (www.mathworks.com).

1.4 Areas of Image Processing Covered in the Book

Every chapter in the book contains the pertinent MATLAB and Image Processing Toolbox material needed to implement the image processing methods discussed. When a MATLAB or toolbox function does not exist to implement a specific method, a custom function is developed and documented. As noted earlier, a complete listing of every new function is available. The remaining twelve chapters cover material in the following areas.

Chapter 2: Fundamentals. This chapter covers the fundamentals of MATLAB notation, matrix indexing, and programming concepts. This material serves as foundation for the rest of the book.

Chapter 3: Intensity Transformations and Spatial Filtering. This chapter covers in detail how to use MATLAB and the Image Processing Toolbox to implement intensity transformation functions. Linear and nonlinear spatial filters are covered and illustrated in detail. We also develop a set of basic functions for fuzzy intensity transformations and spatial filtering.

Chapter 4: Processing in the Frequency Domain. The material in this chapter shows how to use toolbox functions for computing the forward and inverse 2-D fast Fourier transforms (FFTs), how to visualize the Fourier spectrum, and how to implement filtering in the frequency domain. Shown also is a method for generating frequency domain filters from specified spatial filters.

Chapter 5: Image Restoration. Traditional linear restoration methods, such as the Wiener filter, are covered in this chapter. Iterative, nonlinear methods, such as the Richardson-Lucy method and maximum-likelihood estimation for blind deconvolution, are discussed and illustrated. Image reconstruction from projections and how it is used in computed tomography are discussed also in this chapter.

Chapter 6: Geometric Transformations and Image Registration. This chapter discusses basic forms and implementation techniques for geometric image transformations, such as affine and projective transformations. Interpolation methods are presented also. Different image registration techniques are discussed, and several examples of transformation, registration, and visualization methods are given.

Chapter 7: Color Image Processing. This chapter deals with pseudocolor and full-color image processing. Color models applicable to digital image processing are discussed, and Image Processing Toolbox functionality in color processing is extended with additional color models. The chapter also covers applications of color to edge detection and region segmentation.

Chapter 8: Wavelets. The Image Processing Toolbox does not have wavelet transform functions. Although the MathWorks offers a Wavelet Toolbox, we develop in this chapter an independent set of wavelet transform functions that allow implementation all the wavelet-transform concepts discussed in Chapter 7 of *Digital Image Processing* by Gonzalez and Woods.

Chapter 9: Image Compression. The toolbox does not have any data compression functions. In this chapter, we develop a set of functions that can be used for this purpose.

Chapter 10: Morphological Image Processing. The broad spectrum of functions available in toolbox for morphological image processing are explained and illustrated in this chapter using both binary and gray-scale images.

Chapter 11: Image Segmentation. The set of toolbox functions available for image segmentation are explained and illustrated in this chapter. Functions for Hough transform processing are discussed, and custom region growing and thresholding functions are developed.

Chapter 12: Representation and Description. Several new functions for object representation and description, including chain-code and polygonal representations, are developed in this chapter. New functions are included also for object description, including Fourier descriptors, texture, and moment invariants. These functions complement an extensive set of region property functions available in the Image Processing Toolbox.

Chapter 13: Object Recognition. One of the important features of this chapter is the efficient implementation of functions for computing the Euclidean and Mahalanobis distances. These functions play a central role in pattern matching. The chapter also contains a comprehensive discussion on how to manipulate strings of symbols in MATLAB. String manipulation and matching are important in structural pattern recognition.

In addition to the preceding material, the book contains three appendices.

Appendix A: This appendix summarizes Image Processing Toolbox and custom image-processing functions developed in the book. Relevant MATLAB functions also are included. This is a useful reference that provides a global overview of all functions in the toolbox and the book.

Appendix B: Implementation of graphical user interfaces (GUIs) in MATLAB are discussed in this appendix. GUIs complement the material in the book because they simplify and make more intuitive the control of interactive functions.

Appendix C: The code for many custom functions is included in the body of the text at the time the functions are developed. Some function listings are deferred to this appendix when their inclusion in the main text would break the flow of explanations.

1.5 The Book Web Site

An important feature of this book is the support contained in the book web site. The site address is

www.ImageProcessingPlace.com

This site provides support to the book in the following areas:

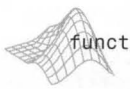
- Availability of M-files, including executable versions of all M-files in the book
- Tutorials
- Projects
- Teaching materials
- Links to databases, including all images in the book
- Book updates
- Background publications

The same site also supports the Gonzalez-Woods book and thus offers complementary support on instructional and research topics.

1.6 Notation

Equations in the book are typeset using familiar italic and Greek symbols, as in $f(x, y) = A\sin(ux + vy)$ and $\phi(u, v) = \tan^{-1}\left[I(u, v)/R(u, v)\right]$. All MATLAB function names and symbols are typeset in monospace font, as in `fft2(f)`, `logical(A)`, and `roipoly(f, c, r)`.

The first occurrence of a MATLAB or Image Processing Toolbox function is highlighted by use of the following icon on the page margin:



Similarly, the first occurrence of a new (custom) function developed in the book is highlighted by use of the following icon on the page margin:

function name

The symbol ▬▬ is used as a visual cue to denote the end of a function listing.

When referring to keyboard keys, we use bold letters, such as **Return** and **Tab**. We also use bold letters when referring to items on a computer screen or menu, such as **File** and **Edit**.

1.7 The MATLAB Desktop

The *MATLAB Desktop* is the main working environment. It is a set of graphics tools for tasks such as running MATLAB commands, viewing output, editing and managing files and variables, and viewing session histories. Figure 1.1 shows the MATLAB Desktop in the default configuration. The Desktop com-

ponents shown are the Command Window, the Workspace Browser, the Current Directory Browser, and the Command History Window. Figure 1.1 also shows a Figure Window, which is used to display images and graphics.

The *Command Window* is where the user types MATLAB commands at the prompt (>>). For example, a user can call a MATLAB function, or assign a value to a variable. The set of variables created in a session is called the *Workspace*, and their values and properties can be viewed in the *Workspace Browser*.

Directories are called *folders* in Windows.

The top-most rectangular window shows the user's *Current Directory*, which typically contains the path to the files on which a user is working at a given time. The current directory can be changed using the arrow or browse button ("...") to the right of the *Current Directory Field*. Files in the Current Directory can be viewed and manipulated using the *Current Directory Browser*.

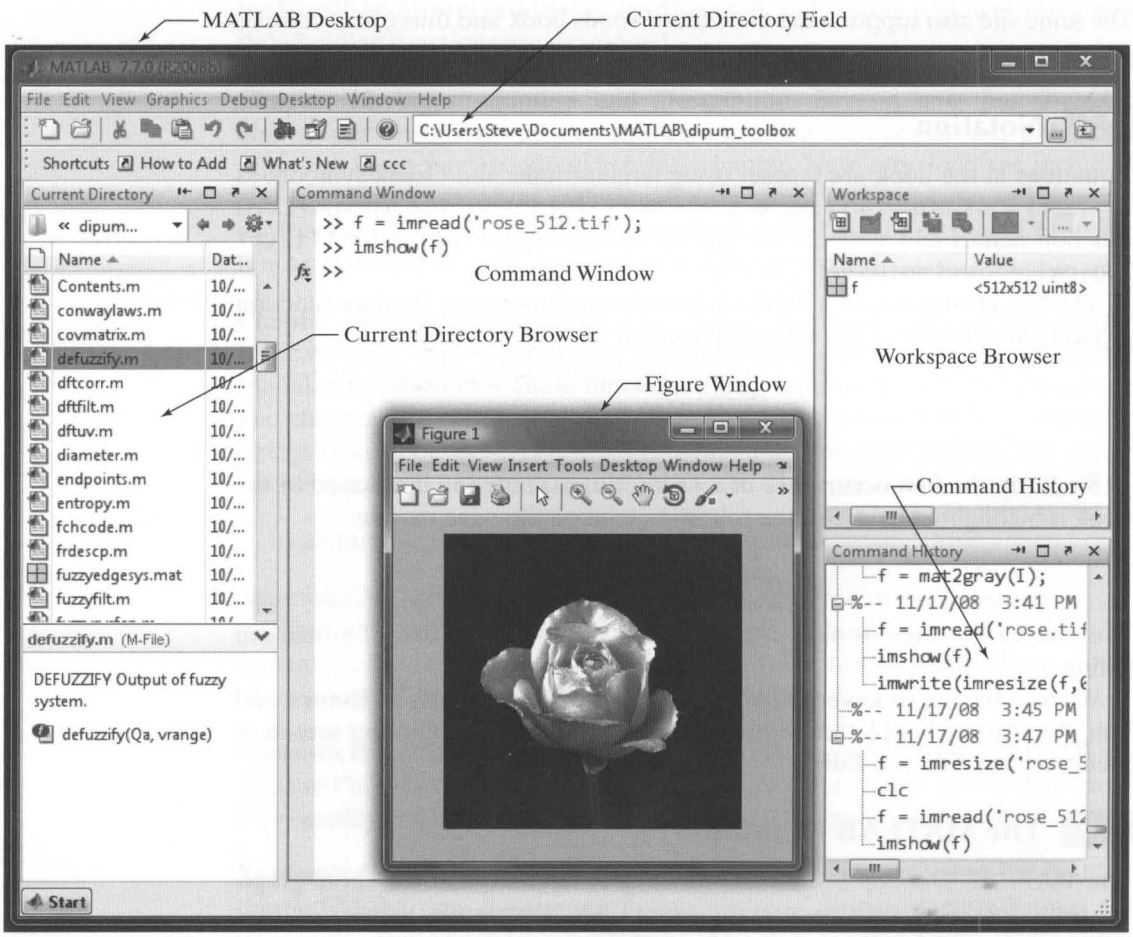


FIGURE 1.1 The MATLAB Desktop with its typical components.

The *Command History Window* displays a log of MATLAB statements executed in the Command Window. The log includes both current and previous sessions. From the Command History Window a user can right-click on previous statements to copy them, re-execute them, or save them to a file. These features are useful for experimenting with various commands in a work session, or for reproducing work performed in previous sessions.

The MATLAB Desktop may be configured to show one, several, or all these tools, and favorite Desktop layouts can be saved for future use. Table 1.1 summarizes all the available Desktop tools.

MATLAB uses a *search path* to find M-files and other MATLAB-related files, which are organized in directories in the computer file system. Any file run in MATLAB must reside in the Current Directory or in a directory that is on the search path. By default, the files supplied with MATLAB and MathWorks toolboxes are included in the search path. The easiest way to see which directories are on the search path, or to add or modify a search path, is to select **Set Path** from the **File** menu on the desktop, and then use the **Set Path** dialog box. It is good practice to add commonly used directories to the search path to avoid repeatedly having to browse to the location of these directories.

Typing `clear` at the prompt removes all variables from the workspace. This frees up system memory. Similarly, typing `clc` clears the contents of the command window. See the help page for other uses and syntax forms.



TABLE 1.1 MATLAB desktop tools.

Tool	Description
Array Editor	View and edit array contents.
Command History Window	View a log of statements entered in the Command Window; search for previously executed statements, copy them, and re-execute them.
Command Window	Run MATLAB statements.
Current Directory Browser	View and manipulate files in the current directory.
Current Directory Field	Shows the path leading to the current directory.
Editor/Debugger	Create, edit, debug, and analyze M-files.
Figure Windows	Display, modify, annotate, and print MATLAB graphics.
File Comparisons	View differences between two files.
Help Browser	View and search product documentation.
Profiler	Measure execution time of MATLAB functions and lines; count how many times code lines are executed.
Start Button	Run product tools and access product documentation and demos.
Web Browser	View HTML and related files produced by MATLAB or other sources.
Workspace Browser	View and modify contents of the workspace.

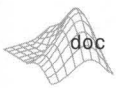
1.7.1 Using the MATLAB Editor/Debugger

The *MATLAB Editor/Debugger* (or just the *Editor*) is one of the most important and versatile of the Desktop tools. Its primary purpose is to create and edit MATLAB function and script files. These files are called *M-files* because their filenames use the extension `.m`, as in `pixeldup.m`. The Editor highlights different MATLAB code elements in color; also, it analyzes code to offer suggestions for improvements. The Editor is the tool of choice for working with M-files. With the Editor, a user can set debugging breakpoints, inspect variables during code execution, and step through code lines. Finally, the Editor can publish MATLAB M-files and generate output to formats such as HTML, LaTeX, Word, and PowerPoint.

To open the editor, type `edit` at the prompt in the Command Window. Similarly, typing `edit filename` at the prompt opens the M-file `filename.m` in an editor window, ready for editing. The file must be in the current directory, or in a directory in the search path.

1.7.2 Getting Help

The principal way to get help is to use the MATLAB *Help Browser*, opened as a separate window either by clicking on the question mark symbol (**?**) on the desktop toolbar, or by typing `doc` (one word) at the prompt in the Command Window. The Help Browser consists of two panes, the *help navigator pane*, used to find information, and the *display pane*, used to view the information. Self-explanatory tabs on the navigator pane are used to perform a search. For example, help on a specific function is obtained by selecting the **Search** tab and then typing the function name in the **Search for** field. It is good practice to open the Help Browser at the beginning of a MATLAB session to have help readily available during code development and other MATLAB tasks.



Another way to obtain help for a specific function is by typing `doc` followed by the function name at the command prompt. For example, typing `doc file_name` displays the reference page for the function called `file_name` in the display pane of the Help Browser. This command opens the browser if it is not open already. The `doc` function works also for user-written M-files that contain help text. See Section 2.10.1 for an explanation of M-file help text.

When we introduce MATLAB and Image Processing Toolbox functions in the following chapters, we often give only representative syntax forms and descriptions. This is necessary either because of space limitations or to avoid deviating from a particular discussion more than is absolutely necessary. In these cases we simply introduce the syntax required to execute the function in the form required at that point in the discussion. By being comfortable with MATLAB documentation tools, you can then explore a function of interest in more detail with little effort.

Finally, the MathWorks' web site mentioned in Section 1.3 contains a large database of help material, contributed functions, and other resources that

should be utilized when the local documentation contains insufficient information about a desired topic. Consult the book web site (see Section 1.5) for additional MATLAB and M-function resources.

1.7.3 Saving and Retrieving Work Session Data

There are several ways to save or load an entire work session (the contents of the Workspace Browser) or selected workspace variables in MATLAB. The simplest is as follows: To save the entire workspace, right-click on any blank space in the Workspace Browser window and select **Save Workspace As** from the menu that appears. This opens a directory window that allows naming the file and selecting any folder in the system in which to save it. Then click **Save**. To save a selected variable from the Workspace, select the variable with a left click and right-click on the highlighted area. Then select **Save Selection As** from the menu that appears. This opens a window from which a folder can be selected to save the variable. To select multiple variables, use shift-click or control-click in the familiar manner, and then use the procedure just described for a single variable. All files are saved in a binary format with the extension `.mat`. These saved files commonly are referred to as *MAT-files*, as indicated earlier. For example, a session named, say, mywork_2009_02_10, would appear as the MAT-file mywork_2009_02_10.mat when saved. Similarly, a saved image called `final_image` (which is a single variable in the workspace) will appear when saved as final_image.mat.

To load saved workspaces and/or variables, left-click on the folder icon on the toolbar of the Workspace Browser window. This causes a window to open from which a folder containing the MAT-files of interest can be selected. Double-clicking on a selected MAT-file or selecting **Open** causes the contents of the file to be restored in the Workspace Browser window.

It is possible to achieve the same results described in the preceding paragraphs by typing `save` and `load` at the prompt, with the appropriate names and path information. This approach is not as convenient, but it is used when formats other than those available in the menu method are required. Functions `save` and `load` are useful also for writing M-files that save and load workspace variables. As an exercise, you are encouraged to use the Help Browser to learn more about these two functions.

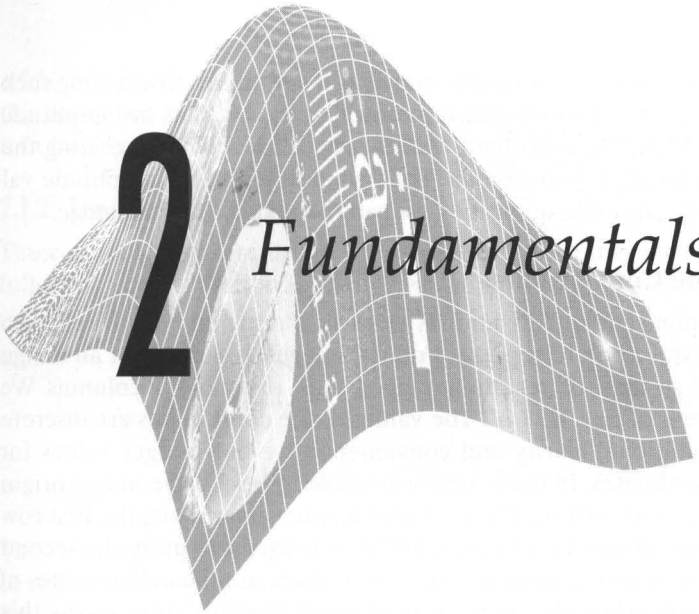


1.8 How References Are Organized in the Book

All references in the book are listed in the Bibliography by author and date, as in Soille [2003]. Most of the background references for the theoretical content of the book are from Gonzalez and Woods [2008]. In cases where this is not true, the appropriate new references are identified at the point in the discussion where they are needed. References that are applicable to all chapters, such as MATLAB manuals and other general MATLAB references, are so identified in the Bibliography.

Summary

In addition to a brief introduction to notation and basic MATLAB tools, the material in this chapter emphasizes the importance of a comprehensive prototyping environment in the solution of digital image processing problems. In the following chapter we begin to lay the foundation needed to understand Image Processing Toolbox functions and introduce a set of fundamental programming concepts that are used throughout the book. The material in Chapters 3 through 13 spans a wide cross section of topics that are in the mainstream of digital image processing applications. However, although the topics covered are varied, the discussion in those chapters follows the same basic theme of demonstrating how combining MATLAB and toolbox functions with new code can be used to solve a broad spectrum of image-processing problems.



2 Fundamentals

Preview

As mentioned in the previous chapter, the power that MATLAB brings to digital image processing is an extensive set of functions for processing multidimensional arrays of which images (two-dimensional numerical arrays) are a special case. The Image Processing Toolbox is a collection of functions that extend the capability of the MATLAB numeric computing environment. These functions, and the expressiveness of the MATLAB language, make image-processing operations easy to write in a compact, clear manner, thus providing an ideal software prototyping environment for the solution of image processing problems. In this chapter we introduce the basics of MATLAB notation, discuss a number of fundamental toolbox properties and functions, and begin a discussion of programming concepts. Thus, the material in this chapter is the foundation for most of the software-related discussions in the remainder of the book.

2.1 Digital Image Representation

An image may be defined as a two-dimensional function $f(x, y)$, where x and y are *spatial* (plane) *coordinates*, and the amplitude of f at any pair of coordinates is called the *intensity* of the image at that point. The term *gray level* is used often to refer to the intensity of monochrome images. Color images are formed by a combination of individual images. For example, in the RGB color system a color image consists of three individual monochrome images, referred to as the *red* (R), *green* (G), and *blue* (B) *primary* (or *component*) *images*. For this reason, many of the techniques developed for monochrome images can be extended to color images by processing the three component images individually. Color image processing is the topic of Chapter 7. An image may be continuous

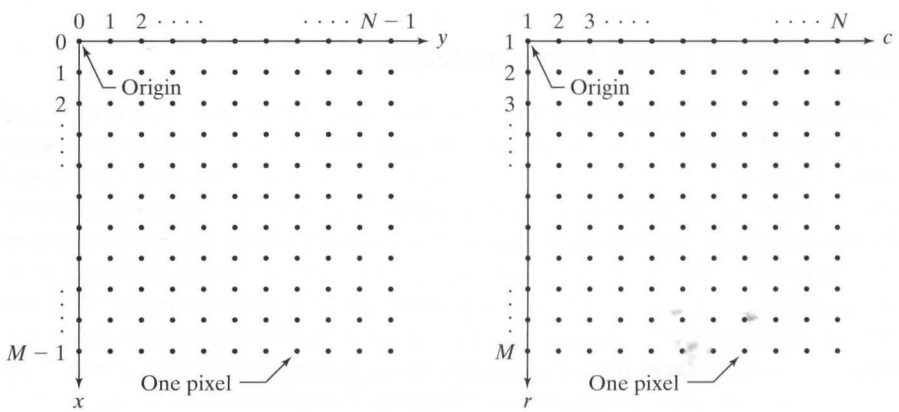
with respect to the x- and y-coordinates, and also in amplitude. Converting such an image to digital form requires that the coordinates, as well as the amplitude, be digitized. Digitizing the coordinate values is called *sampling*; digitizing the amplitude values is called *quantization*. Thus, when x, y, and the amplitude values of f are all finite, discrete quantities, we call the image a *digital image*.

2.1.1 Coordinate Conventions

The result of sampling and quantization is a matrix of real numbers. We use two principal ways in this book to represent digital images. Assume that an image $f(x, y)$ is sampled so that the resulting image has M rows and N columns. We say that the image is of *size* $M \times N$. The values of the coordinates are discrete quantities. For notational clarity and convenience, we use integer values for these discrete coordinates. In many image processing books, the image origin is defined to be at $(x, y) = (0, 0)$. The next coordinate values along the first row of the image are $(x, y) = (0, 1)$. The notation $(0, 1)$ is used to signify the second sample along the first row. It *does not* mean that these are the *actual* values of physical coordinates when the image was sampled. Figure 2.1(a) shows this coordinate convention. Note that x ranges from 0 to $M - 1$ and y from 0 to $N - 1$ in integer increments.

The coordinate convention used in the Image Processing Toolbox to denote arrays is different from the preceding paragraph in two minor ways. First, instead of using (x, y), the toolbox uses the notation (r, c) to indicate rows and columns. Note, however, that the order of coordinates is the same as the order discussed in the previous paragraph, in the sense that the first element of a coordinate tuple, (a, b), refers to a row and the second to a column. The other difference is that the origin of the coordinate system is at $(r, c) = (1, 1)$; thus, r ranges from 1 to M, and c from 1 to N, in integer increments. Figure 2.1(b) illustrates this coordinate convention.

Image Processing Toolbox documentation refers to the coordinates in Fig. 2.1(b) as *pixel coordinates*. Less frequently, the toolbox also employs another coordinate convention, called *spatial coordinates*, that uses x to refer to columns and y to refers to rows. This is the opposite of our use of variables x and y. With



a b

FIGURE 2.1 Coordinate conventions used (a) in many image processing books, and (b) in the Image Processing Toolbox.

a few exceptions, we do not use the toolbox's spatial coordinate convention in this book, but many MATLAB functions do, and you will definitely encounter it in toolbox and MATLAB documentation.

2.1.2 Images as Matrices

The coordinate system in Fig. 2.1(a) and the preceding discussion lead to the following representation for a digitized image:

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix}$$

The right side of this equation is a digital image by definition. Each element of this array is called an *image element*, *picture element*, *pixel*, or *pel*. The terms *image* and *pixel* are used throughout the rest of our discussions to denote a digital image and its elements.

A digital image can be represented as a MATLAB matrix:

$$\mathtt{f} = \begin{bmatrix} \mathtt{f(1,1)} & \mathtt{f(1,2)} & \cdots & \mathtt{f(1,N)} \\ \mathtt{f(2,1)} & \mathtt{f(2,2)} & \cdots & \mathtt{f(2,N)} \\ \vdots & \vdots & & \vdots \\ \mathtt{f(M,1)} & \mathtt{f(M,2)} & \cdots & \mathtt{f(M,N)} \end{bmatrix}$$

MATLAB documentation uses the terms *matrix* and *array* interchangeably. However, keep in mind that a matrix is two dimensional, whereas an array can have any finite dimension.

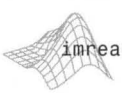
where `f(1, 1)` = $f(0,0)$ (note the use of a monospace font to denote MATLAB quantities). Clearly, the two representations are identical, except for the shift in origin. The notation `f(p, q)` denotes the element located in row `p` and column `q`. For example, `f(6, 2)` is the element in the sixth row and second column of matrix `f`. Typically, we use the letters `M` and `N`, respectively, to denote the number of rows and columns in a matrix. A `1`×`N` matrix is called a *row vector*, whereas an `M`×`1` matrix is called a *column vector*. A `1`×`1` matrix is a *scalar*.

Matrices in MATLAB are stored in variables with names such as `A`, `a`, `RGB`, `real_array`, and so on. Variables must begin with a letter and contain only letters, numerals, and underscores. As noted in the previous paragraph, all MATLAB quantities in this book are written using monospace characters. We use conventional Roman, italic notation, such as $f(x,y)$, for mathematical expressions.

2.2 Reading Images

Recall from Section 1.6 that we use margin icons to highlight the first use of a MATLAB or toolbox function.

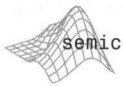
Images are read into the MATLAB environment using function `imread`, whose basic syntax is




```
imread('filename')
```

Here, `filename` is a string containing the complete name of the image file (including any applicable extension). For example, the statement

```
>> f = imread('chestxray.jpg');
```



reads the image from the JPEG file `chestxray` into image array `f`. Note the use of single quotes (`'`) to delimit the string `filename`. The semicolon at the end of a statement is used by MATLAB for *suppressing* output. If a semicolon is not included, MATLAB displays on the screen the results of the operation(s) specified in that line. The prompt symbol (`>>`) designates the beginning of a command line, as it appears in the MATLAB Command Window (see Fig. 1.1).



When, as in the preceding command line, no path information is included in `filename`, `imread` reads the file from the Current Directory and, if that fails, it tries to find the file in the MATLAB search path (see Section 1.7). The simplest way to read an image from a specified directory is to include a full or relative path to that directory in `filename`. For example,

In Windows, directories are called *folders*.

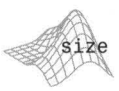
```
>> f = imread('D:\myimages\chestxray.jpg');
```

reads the image from a directory called `myimages` in the `D:` drive, whereas

```
>> f = imread('.\myimages\chestxray.jpg');
```

reads the image from the `myimages` subdirectory of the current working directory. The MATLAB Desktop displays the path to the Current Directory on the toolbar, which provides an easy way to change it. Table 2.1 lists some of the most popular image/graphics formats supported by `imread` and `imwrite` (`imwrite` is discussed in Section 2.4).

Typing `size` at the prompt gives the row and column dimensions of an image:




```
>> size(f)
ans =
    1024    1024
```

More generally, for an array `A` having an arbitrary number of dimensions, a statement of the form

```
[D1, D2, ..., DK] = size(A)
```

returns the sizes of the first `K` dimensions of `A`. This function is particularly useful in programming to determine automatically the size of a 2-D image:

```
>> [M, N] = size(f);
```

This syntax returns the number of rows (`M`) and columns (`N`) in the image. Similarly, the command

Format Name	Description	Recognized Extensions
BMP†	Windows Bitmap	`.bmp`
CUR	Windows Cursor Resources	`.cur`
FITS†	Flexible Image Transport System	`.fts, .fits`
GIF	Graphics Interchange Format	`.gif`
HDF	Hierarchical Data Format	`.hdf`
ICO†	Windows Icon Resources	`.ico`
JPEG	Joint Photographic Experts Group	`.jpg, .jpeg`
JPEG 2000†	Joint Photographic Experts Group	`.jp2, .jpf, .jpx, j2c, j2k`
PBM	Portable Bitmap	`.pbm`
PGM	Portable Graymap	`.pgm`
PNG	Portable Network Graphics	`.png`
PNM	Portable Any Map	`.pnm`
RAS	Sun Raster	`.ras`
TIFF	Tagged Image File Format	`.tif, .tiff`
XWD	X Window Dump	`.xwd`

TABLE 2.1 Some of the image/graphics formats supported by `imread` and `imwrite`, starting with MATLAB 7.6. Earlier versions support a subset of these formats. See the MATLAB documentation for a complete list of supported formats.

†Supported by `imread`, but not by `imwrite`

```
>> M = size(f, 1);
```

gives the size of `f` along its first dimension, which is defined by MATLAB as the vertical dimension. That is, this command gives the number of rows in `f`. The second dimension of an array is in the horizontal direction, so the statement `size(f, 2)` gives the number of columns in `f`. A *singleton dimension* is any dimension, `dim`, for which `size(A, dim) = 1`.

The `whos` function displays additional information about an array. For instance, the statement

```
>> whos f
```



gives

```
  Name      Size              Bytes  Class     Attributes
  f         1024x1024       1048576  uint8
```

Although not applicable in this example, attributes that might appear under `Attributes` include terms such as `global`, `complex`, and `sparse`.

The Workspace Browser in the MATLAB Desktop displays similar information. The `uint8` entry shown refers to one of several MATLAB data classes discussed in Section 2.5. A semicolon at the end of a `whos` line has no effect, so normally one is not used.

2.3 Displaying Images

Images are displayed on the MATLAB desktop using function `imshow`, which has the basic syntax:



Function `imshow` has a number of other syntax forms for performing tasks such as controlling image magnification. Consult the help page for `imshow` for additional details.

```
imshow(f)
```

where `f` is an image array. Using the syntax

```
imshow(f, [low high])
```

displays as black all values less than or equal to `low`, and as white all values greater than or equal to `high`. The values in between are displayed as intermediate intensity values. Finally, the syntax

```
imshow(f, [ ])
```

sets variable `low` to the minimum value of array `f` and `high` to its maximum value. This form of `imshow` is useful for displaying images that have a low dynamic range or that have positive and negative values.

EXAMPLE 2.1: Reading and displaying images.

■ The following statements read from disk an image called `rose_512.tif`, extract information about the image, and display it using `imshow`:

```
>> f = imread('rose_512.tif');
>> whos f
  Name      Size            Bytes    Class     Attributes
  f         512x512         262144   uint8 array

>> imshow(f)
```

A semicolon at the end of an `imshow` line has no effect, so normally one is not used. Figure 2.2 shows what the output looks like on the screen. The figure

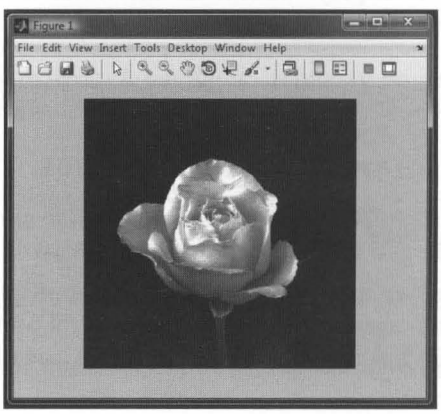
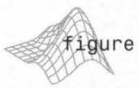


FIGURE 2.2 Screen capture showing how an image appears on the MATLAB desktop. Note the figure number on the top, left of the window. In most of the examples throughout the book, only the images themselves are shown.

number appears on the top, left of the window. Note the various pull-down menus and utility buttons. They are used for processes such as scaling, saving, and exporting the contents of the display window. In particular, the **Edit** menu has functions for editing and formatting the contents before they are printed or saved to disk.

If another image, g, is displayed using `imshow`, MATLAB replaces the image in the figure window with the new image. To keep the first image and output a second image, use function `figure`, as follows:

```
>> figure, imshow(g)
```



Function `figure` creates a figure window. When used without an argument, as shown here, it simply creates a *new* figure window. Typing `figure(n)` forces figure number `n` to become visible.

Using the statement

```
>> imshow(f), figure, imshow(g)
```

displays both images. Note that more than one command can be written on a line, provided that different commands are delimited by commas or semicolons. As mentioned earlier, a semicolon is used whenever it is desired to suppress screen outputs from a command line.

Finally, suppose that we have just read an image, h, and find that using `imshow(h)` produces the image in Fig. 2.3(a). This image has a low dynamic range, a condition that can be remedied for display purposes by using the statement

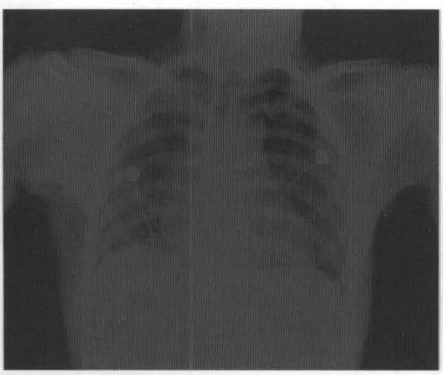
```
>> imshow(h, [ ])
```

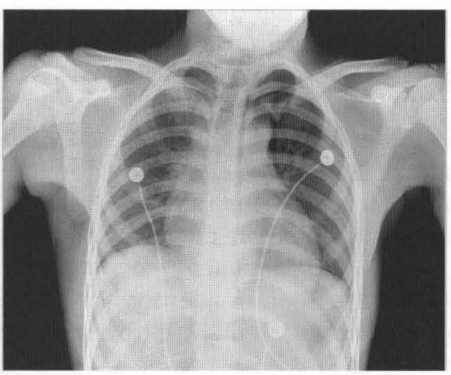
Figure 2.3(b) shows the result. The improvement is apparent. ■

The *Image Tool* in the Image Processing Toolbox provides a more interactive environment for viewing and navigating within images, displaying detailed information about pixel values, measuring distances, and other useful operations. To start the Image Tool, use the `imtool` function. For example, the following statements read an image from a file and then display it using `imtool`:

```
>> f = imread('rose_1024.tif');
>> imtool(f)
```







a b

FIGURE 2.3 (a) An image, h, with low dynamic range. (b) Result of scaling by using `imshow(h, [])`. (Original image courtesy of Dr. David R. Pickens, Vanderbilt University Medical Center.)

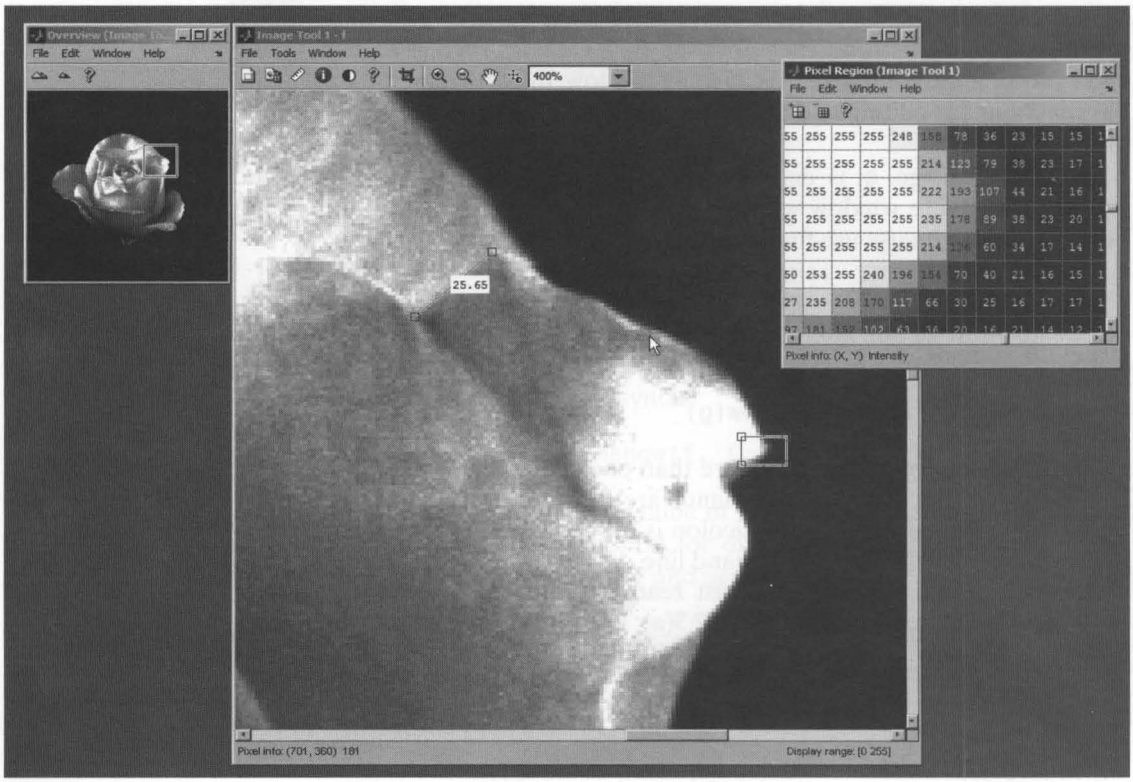


FIGURE 2.4 The Image Tool. The Overview Window, Main Window, and Pixel Region tools are shown.

Figure 2.4 shows some of the windows that might appear when using the Image Tool. The large, central window is the main view. In the figure, it is showing the image pixels at 400% magnification, meaning that each image pixel is rendered on a 4×4 block of screen pixels. The status text at the bottom of the main window shows the column/row location (701, 360) and value (181) of the pixel lying under the mouse cursor (the origin of the image is at the top, left). The *Measure Distance* tool is in use, showing that the distance between the two pixels enclosed by the small boxes is 25.65 units.

The *Overview Window*, on the left side of Fig. 2.4, shows the entire image in a thumbnail view. The *Main Window* view can be adjusted by dragging the rectangle in the Overview Window. The *Pixel Region Window* shows individual pixels from the small square region on the upper right tip of the rose, zoomed large enough to see the actual pixel values.

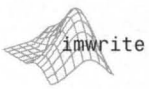
Table 2.2 summarizes the various tools and capabilities associated with the Image Tool. In addition to the these tools, the Main and Overview Window toolbars provide controls for tasks such as image zooming, panning, and scrolling.

Tool	Description
Pixel Information	Displays information about the pixel under the mouse pointer.
Pixel Region	Superimposes pixel values on a zoomed-in pixel view.
Distance	Measures the distance between two pixels.
Image Information	Displays information about images and image files.
Adjust Contrast	Adjusts the contrast of the displayed image.
Crop Image	Defines a crop region and crops the image.
Display Range	Shows the display range of the image data.
Overview	Shows the currently visible image.

TABLE 2.2 Tools associated with the Image Tool.

2.4 Writing Images

Images are written to the Current Directory using function `imwrite`, which has the following basic syntax:




```
imwrite(f, 'filename')
```

With this syntax, the string contained in `filename` must include a recognized file format extension (see Table 2.1). For example, the following command writes `f` to a file called `patient10_run1.tif`:

```
>> imwrite(f, 'patient10_run1.tif')
```

Function `imwrite` writes the image as a TIFF file because it recognizes the `.tif` extension in the filename.

Alternatively, the desired format can be specified explicitly with a third input argument. This syntax is useful when the desired file does not use one of the recognized file extensions. For example, the following command writes `f` to a TIFF file called `patient10.run1`:

```
>> imwrite(f, 'patient10.run1', 'tif')
```

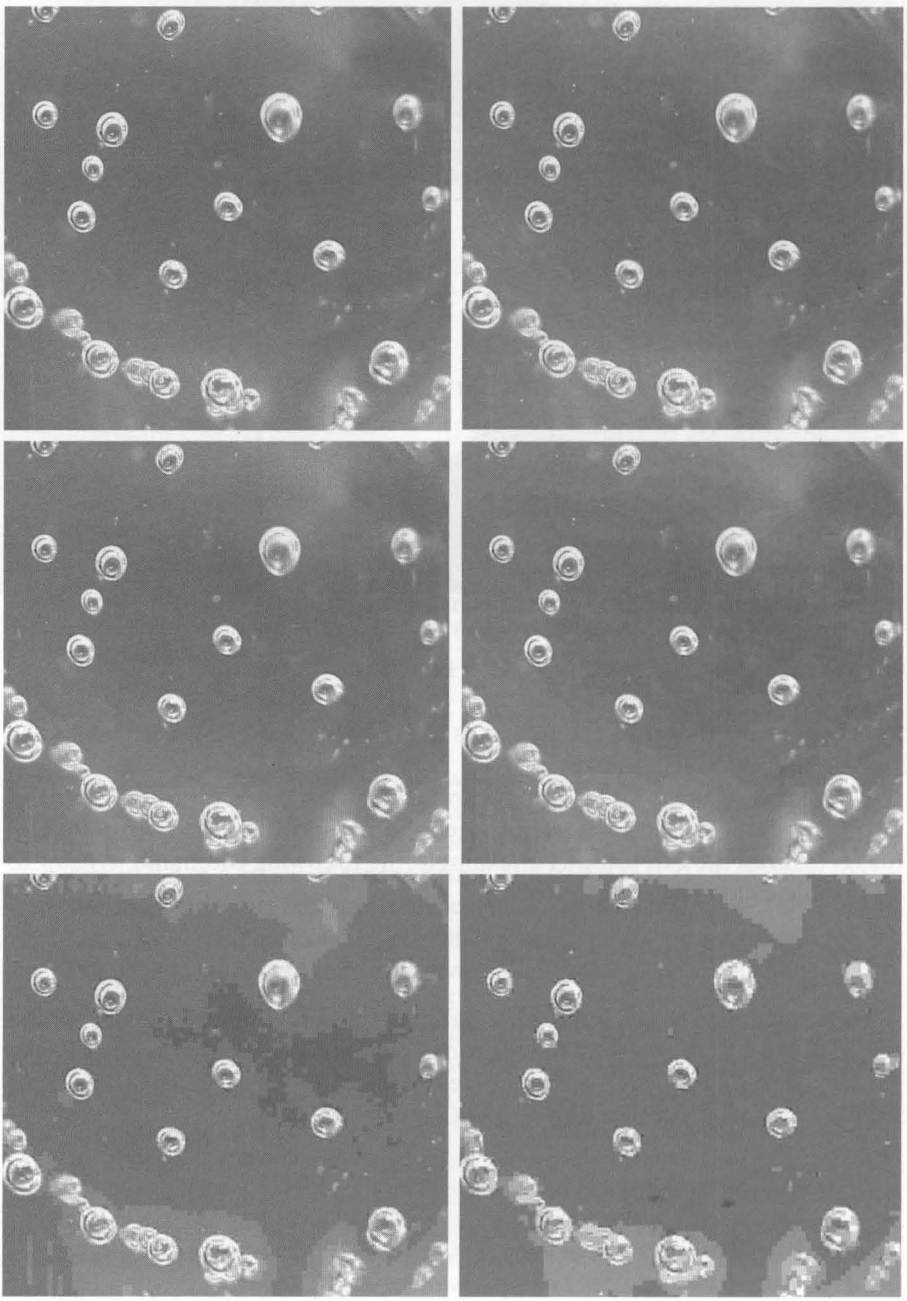
Function `imwrite` can have other parameters, depending on the file format selected. Most of the work in the following chapters deals either with JPEG or TIFF images, so we focus attention here on these two formats. A more general `imwrite` syntax applicable only to JPEG images is

```
imwrite(f, 'filename.jpg', 'quality', q)
```

where `q` is an integer between 0 and 100 (the lower the number the higher the degradation due to JPEG compression).

EXAMPLE 2.2: Writing an image and using function `imfinfo`.

■ Figure 2.5(a) shows an image, `f`, typical of sequences of images resulting from a given chemical process. It is desired to transmit these images on a routine basis to a central site for visual and/or automated inspection. In order to reduce storage requirements and transmission time, it is important that the images be compressed as much as possible, while not degrading their visual



a b
c d
e f

FIGURE 2.5
(a) Original image. (b) through (f) Results of using `jpg` quality values `q = 50, 25, 15, 5,` and `0`, respectively. False contouring begins to be noticeable for `q = 15` [image (d)] and is quite visible for `q = 5` and `q = 0`.

See Example 2.11 for a function that creates all the images in Fig. 2.5 using a loop.

appearance beyond a reasonable level. In this case "reasonable" means no perceptible *false contouring*. Figures 2.5(b) through (f) show the results obtained by writing image `f` to disk (in JPEG format), with `q = 50`, `25`, `15`, `5`, and `0`, respectively. For example, the applicable syntax for `q = 25` is

```
>> imwrite(f, 'bubbles25.jpg', 'quality', 25)
```

The image for `q = 15` [Fig. 2.5(d)] has false contouring that is barely visible, but this effect becomes quite pronounced for `q = 5` and `q = 0`. Thus, an acceptable solution with some margin for error is to compress the images with `q = 25`. In order to get an idea of the compression achieved and to obtain other image file details, we can use function `imfinfo`, which has the syntax

```
imfinfo filename
```



where `filename` is the file name of the image stored on disk. For example,

```
>> imfinfo bubbles25.jpg
```

outputs the following information (note that some fields contain no information in this case):

Recent versions of MATLAB may show more information in the output of `imfinfo`, particularly for images captures using digital cameras.

```
          Filename:  'bubbles25.jpg'
       FileModDate:  '04-Jan-2003 12:31:26'
          FileSize:  13849
            Format:  'jpg'
     FormatVersion:  ''
             Width:  714
            Height:  682
          BitDepth:  8
         ColorType:  'grayscale'
   FormatSignature:  ''
           Comment:  {}
```

where `FileSize` is in bytes. The number of bytes in the original image is computed by multiplying `Width` by `Height` by `BitDepth` and dividing the result by 8. The result is 486948. Dividing this by `FileSize` gives the compression ratio: (486948/13849) = 35.16. This compression ratio was achieved while maintaining image quality consistent with the requirements of the application. In addition to the obvious advantages in storage space, this reduction allows the transmission of approximately 35 times the amount of uncompressed data per unit time.

The information fields displayed by `imfinfo` can be captured into a so-called *structure variable* that can be used for subsequent computations. Using the preceding image as an example, and letting `K` denote the structure variable, we use the syntax

Structures are discussed in Section 2.10.7.

```
>> K = imfinfo('bubbles25.jpg');
```

to store into variable `K` all the information generated by command `imfinfo`.

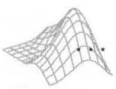
The information generated by `imfinfo` is appended to the structure variable by means of *fields*, separated from `K` by a dot. For example, the image height and width are now stored in structure fields `K.Height` and `K.Width`. As an illustration, consider the following use of structure variable `K` to compute the compression ratio for `bubbles25.jpg`:

```
>> K = imfinfo('bubbles25.jpg');
>> image_bytes = K.Width*K.Height*K.BitDepth/8;
>> compressed_bytes = K.FileSize;
>> compression_ratio = image_bytes/compressed_bytes
compression_ratio =
    35.1612
```

Note that `imfinfo` was used in two different ways. The first was to type `imfinfo bubbles25.jpg` at the prompt, which resulted in the information being displayed on the screen. The second was to type `K=imfinfo('bubbles25.jpg')`, which resulted in the information generated by `imfinfo` being stored in `K`. These two different ways of calling `imfinfo` are an example of *command-function duality*, an important concept that is explained in more detail in the MATLAB documentation. ■

To learn more about command function duality, consult the help page on this topic. (See Section 1.7.2 regarding help pages.)

A more general `imwrite` syntax applicable only to `tif` images has the form



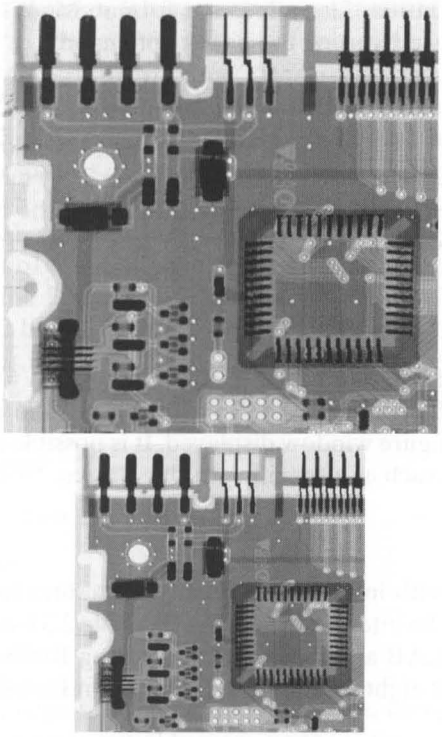
```
imwrite(g, 'filename.tif', 'compression', 'parameter', ...
                           'resolution', [colres rowres])
```

If a statement does not fit on one line, use an ellipsis (three periods), followed by **Return** or **Enter**, to indicate that the statement continues on the next line. There are no spaces between the periods.

where `'parameter'` can have one of the following principal values: `'none'` indicates no compression; `'packbits'` (the default for nonbinary images), `'lwz'`, `'deflate'`, `'jpeg'`, `'ccitt'` (binary images only; the default), `'fax3'` (binary images only), and `'fax4'`. The 1×2 array `[colres rowres]` contains two integers that give the column resolution and row resolution in dots-per-unit (the default values are [72 72]). For example, if the image dimensions are in inches, `colres` is the number of dots (pixels) per inch (dpi) in the vertical direction, and similarly for `rowres` in the horizontal direction. Specifying the resolution by a single scalar, `res`, is equivalent to writing `[res res]`. As you will see in the following example, the TIFF resolution parameter can be used to modify the size of an image in printed documents.

EXAMPLE 2.3: Using `imwrite` parameters.

■ Figure 2.6(a) is an 8-bit X-ray image, `f`, of a circuit board generated during quality inspection. It is in `jpg` format, at 200 dpi. The image is of size 450×450 pixels, so its printed dimensions are 2.25×2.25 inches. We want to store this image in `tif` format, with no compression, under the name `sf`. In addition, we want to reduce the printed size of the image to 1.5×1.5 inches while keeping the pixel count at 450×450. The following statement gives the desired result:

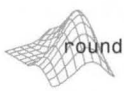


a
b

FIGURE 2.6
Effects of changing the dpi resolution while keeping the number of pixels constant. (a) A 450 × 450 image at 200 dpi (size = 2.25 × 2.25 inches). (b) The same image, but at 300 dpi (size = 1.5 × 1.5 inches). (Original image courtesy of Lixi, Inc.)

```
>> imwrite(f, 'sf.tif', 'compression', 'none', 'resolution', [300 300])
```

The values of the vector `[colres rowres]` were determined by multiplying 200 dpi by the ratio 2.25/1.5 which gives 300 dpi. Rather than do the computation manually, we could write




```
>> res = round(200*2.25/1.5);
>> imwrite(f, 'sf.tif', 'compression', 'none' ,'resolution', res)
```

where function `round` rounds its argument to the nearest integer. It is important to note that the number of pixels was not changed by these commands. Only the printed size of the image changed. The original 450 × 450 image at 200 dpi is of size 2.25 × 2.25 inches. The new 300-dpi image [Fig. 2.6(b)] is identical, except that its 450 × 450 pixels are distributed over a 1.5 × 1.5-inch area. Processes such as this are useful for controlling the size of an image in a printed document without sacrificing resolution. ■

Sometimes, it is necessary to export images and plots to disk the way they appear on the MATLAB desktop. The *contents* of a figure window can be exported to disk in two ways. The first is to use the **File** pull-down menu in the figure window (see Fig. 2.2) and then choose **Save As**. With this option, the

user can select a location, file name, and format. More control over export parameters is obtained by using the `print` command:




```
print -fno -dfileformat -rresno filename
```

where *no* refers to the figure number in the figure window of interest, *fileformat* refers to one of the file formats in Table 2.1, *resno* is the resolution in dpi, and `filename` is the name we wish to assign the file. For example, to export the contents of the figure window in Fig. 2.2 as a `tif` file at 300 dpi, and under the name `hi_res_rose`, we would type

```
>> print -f1 -dtiff -r300 hi_res_rose
```

This command sends the file `hi_res_rose.tif` to the Current Directory. If we type `print` at the prompt, MATLAB prints (to the default printer) the contents of the last figure window displayed. It is possible also to specify other options with `print`, such as a specific printing device.

2.5 Classes

Although we work with integer coordinates, the values (intensities) of pixels are not restricted to be integers in MATLAB. Table 2.3 lists the various *classes* supported by MATLAB and the Image Processing Toolbox† for representing pixel values. The first eight entries in the table are referred to as *numeric* class-

TABLE 2.3
Classes used for image processing in MATLAB. The first eight entries are referred to as *numeric* classes, the ninth entry is the *char* class, and the last entry is the *logical* class.

Name	Description
`double`	Double-precision, floating-point numbers in the approximate range $\pm 10^{308}$ (8 bytes per element).
`single`	Single-precision floating-point numbers with values in the approximate range $\pm 10^{38}$ (4 bytes per element).
`uint8`	Unsigned 8-bit integers in the range [0, 255] (1 byte per element).
`uint16`	Unsigned 16-bit integers in the range [0, 65535] (2 bytes per element).
`uint32`	Unsigned 32-bit integers in the range [0, 4294967295] (4 bytes per element).
`int8`	Signed 8-bit integers in the range [−128, 127] (1 byte per element).
`int16`	Signed 16-bit integers in the range [−32768, 32767] (2 bytes per element).
`int32`	Signed 32-bit integers in the range [−2147483648, 2147483647] (4 bytes per element).
`char`	Characters (2 bytes per element).
`logical`	Values are 0 or 1 (1 byte per element).

† MATLAB supports two other numeric classes not listed in Table 2.3, `uint64` and `int64`. The toolbox does not support these classes, and MATLAB arithmetic support for them is limited.

es. The ninth entry is the *char* (character) class and, as shown, the last entry is the *logical* class.

Classes `uint8` and `logical` are used extensively in image processing, and they are the usual classes encountered when reading images from image file formats such as TIFF or JPEG. These classes use 1 byte to represent each pixel. Some scientific data sources, such as medical imagery, require more dynamic range than is provided by `uint8`, so the `uint16` and `int16` classes are used often for such data. These classes use 2 bytes for each array element. The floating-point classes `double` and `single` are used for computationally intensive operations such as the Fourier transform (see Chapter 4). Double-precision floating-point uses 8 bytes per array element, whereas single-precision floating-point uses 4 bytes. The `int8`, `uint32`, and `int32` classes, although supported by the toolbox, are not used commonly for image processing.

2.6 Image Types

The toolbox supports four types of images:

- Gray-scale images
- Binary images
- Indexed images
- RGB images

Gray-scale images are referred to *as intensity images* in earlier versions of the toolbox. In the book, we use the two terms interchangeably when working with monochrome images.

Most monochrome image processing operations are carried out using binary or gray-scale images, so our initial focus is on these two image types. Indexed and RGB color images are discussed in Chapter 7.

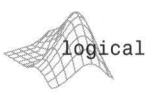
2.6.1 Gray-scale Images

A *gray-scale image* is a data matrix whose values represent shades of gray. When the elements of a gray-scale image are of class `uint8` or `uint16`, they have integer values in the range [0, 255] or [0, 65535], respectively. If the image is of class `double` or `single`, the values are floating-point numbers (see the first two entries in Table 2.3). Values of `double` and `single` gray-scale images normally are scaled in the range [0, 1], although other ranges can be used.

2.6.2 Binary Images

Binary images have a very specific meaning in MATLAB. A *binary image* is a *logical* array of 0s and 1s. Thus, an array of 0s and 1s whose values are of data class, say, `uint8`, is not considered a binary image in MATLAB. A numeric array is converted to binary using function `logical`. Thus, if `A` is a numeric array consisting of 0s and 1s, we create a logical array `B` using the statement

```
B = logical(A)
```



If `A` contains elements other than 0s and 1s, the `logical` function converts all nonzero quantities to logical 1s and all entries with value 0 to logical 0s. Using relational and logical operators (see Section 2.10.2) also results in logical arrays.

To test if an array is of class `logical` we use the `islogical` function:




```
islogical(C)
```

See Table 2.9 for a list of other functions based on the `is...` construct.

If `C` is a logical array, this function returns a 1. Otherwise it returns a 0. Logical arrays can be converted to numeric arrays using the class conversion functions discussed in Section 2.7.

2.6.3 A Note on Terminology

Considerable care was taken in the previous two sections to clarify the use of the terms *class* and *image type*. In general, we refer to an image as being a "`class` image_type image," where `class` is one of the entries from Table 2.3, and image_type is one of the image types defined at the beginning of this section. Thus, an image is characterized by *both* a class *and* a type. For instance, a statement discussing an "`uint8` gray-scale image" is simply referring to a gray-scale image whose pixels are of class `uint8`. Some functions in the toolbox support all the data classes listed in Table 2.3, while others are very specific as to what constitutes a valid class.

2.7 Converting between Classes

Converting images from one class to another is a common operation. When converting between classes, keep in mind the value ranges of the classes being converted (see Table 2.3).

The general syntax for class conversion is

```
B = class_name(A)
```

where `class_name` is one of the names in the first column of Table 2.3. For example, suppose that `A` is an array of class `uint8`. A double-precision array, `B`, is generated by the command `B = double(A)`. If `C` is an array of class `double` in which all values are in the range [0, 255] (but possibly containing fractional values), it can be converted to an `uint8` array with the command `D = uint8(C)`. If an array of class `double` has any values outside the range [0, 255] and it is converted to class `uint8` in the manner just described, MATLAB converts to 0 all values that are less than 0, and converts to 255 all values that are greater than 255. Numbers in between are rounded to the nearest integer. Thus, proper scaling of a `double` array so that its elements are in the range [0, 255] is necessary before converting it to `uint8`. As indicated in Section 2.6.2, converting any of the numeric data classes to `logical` creates an array with logical 1s in locations where the input array has nonzero values, and logical 0s in places where the input array contains 0s.

To simplify terminology, statements referring to values of class `double` are applicable also to the `single` class, unless stated otherwise. Both refer to floating point numbers, the only difference between them being precision and the number of bytes needed for storage.

The toolbox provides specific functions (Table 2.4) that perform the scaling and other bookkeeping necessary to convert images from one class to another. Function `im2uint8`, for example, creates a `unit8` image after detecting the

Name	Converts Input to:	Valid Input Image Data Classes
`im2uint8`	`uint8`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`im2uint16`	`uint16`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`im2double`	`double`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`im2single`	`single`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`mat2gray`	`double` in the range [0, 1]	`logical`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, `single`, and `double`
`im2bw`	`logical`	`uint8`, `uint16`, `int16`, `single`, and `double`

TABLE 2.4 Toolbox functions for converting images from one class to another.

data class of the input and performing all the necessary scaling for the toolbox to recognize the data as valid image data. For example, consider the following image `f` of class `double`, which could be the result of an intermediate computation:

```
f =
   -0.5    0.5
   0.75    1.5
```

Performing the conversion

```
>> g = im2uint8(f)
```



yields the result

```
g =
      0   128
    191   255
```

from which we see that function `im2uint8` sets to 0 all values in the input that are less than 0, sets to 255 all values in the input that are greater than 1, and multiplies all other values by 255. Rounding the results of the multiplication to the nearest integer completes the conversion.



Function `im2double` converts an input to class `double`. If the input is of class `uint8`, `uint16`, or `logical`, function `im2double` converts it to class `double` with values in the range [0, 1]. If the input is of class `single`, or is already of class `double`, `im2double` returns an array that is of class `double`, but is numerically equal to the input. For example, if an array of class `double` results from computations that yield values outside the range [0, 1], inputting this array into

`im2double` will have no effect. As explained below, function `mat2gray` can be used to convert an array of any of the classes in Table 2.4 to a `double` array with values in the range [0, 1].

As an illustration, consider the class `uint8` image

Section 2.8.2 explains the use of square brackets and semicolons to specify matrices.

```
>> h = uint8([25 50; 128 200]);
```

Performing the conversion

```
>> g = im2double(h)
```

yields the result

```
g =
    0.0980    0.1961
    0.4706    0.7843
```

from which we infer that the conversion when the input is of class `uint8` is done simply by dividing each value of the input array by 255. If the input is of class `uint16` the division is by 65535.

Toolbox function `mat2gray` converts an image of any of the classes in Table 2.4 to an array of class `double` *scaled* to the range [0, 1]. The calling syntax is




```
g = mat2gray(A, [Amin, Amax])
```

where image `g` has values in the range 0 (black) to 1 (white). The specified parameters, `Amin` and `Amax`, are such that values less than `Amin` in `A` become 0 in `g`, and values greater than `Amax` in `A` correspond to 1 in `g`. The syntax

```
g = mat2gray(A)
```

sets the values of `Amin` and `Amax` to the actual minimum and maximum values in `A`. The second syntax of `mat2gray` is a very useful tool because it scales the entire range of values in the input to the range [0, 1], independently of the class of the input, thus eliminating clipping.

Finally, we consider conversion to class `logical`. (Recall that the Image Processing Toolbox treats logical matrices as binary images.) Function `logical` converts an input array to a logical array. In the process, nonzero elements in the input are converted to 1s, and 0s are converted to 0s in the output. An alternative conversion procedure that often is more useful is to use a relational operator, such as >, with a threshold value. For example, the syntax

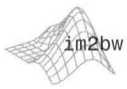
See Section 2.10.2 regarding logical and relational operators.

```
g = f > T
```

produces a logical matrix containing 1s wherever the elements of `f` are greater than `T` and 0s elsewhere.

Toolbox function `im2bw` performs this thresholding operation in a way that automatically scales the specified threshold in different ways, depending on the class of the input image. The syntax is

```
g = im2bw(f, T)
```



Values specified for the threshold `T` must be in the range [0, 1], regardless of the class of the input. The function automatically scales the threshold value according to the input image class. For example, if `f` is `uint8` and `T` is `0.4`, then `im2bw` thresholds the pixels in `f` by comparing them to `255 * 0.4 = 102`.

EXAMPLE 2.4: Converting between image classes.

■ We wish to convert the following small, `double` image

```
>> f = [1 2; 3 4]
f =
     1     2
     3     4
```

to binary, such that values 1 and 2 become 0 and the other two values become 1. First we convert it to the range [0, 1]:

```
>> g = mat2gray(f)
g =
         0    0.3333
    0.6667    1.0000
```

Then we convert it to binary using a threshold, say, of value 0.6:

```
>> gb = im2bw(g, 0.6)
gb =
     0     0
     1     1
```

As mentioned earlier, we can generate a binary array directly using relational operators. Thus we get the same result by writing

```
>> gb = f > 2
gb =
     0     0
     1     1
```

Suppose now that we want to convert `gb` to a numerical array of 0s and 1s of class `double`. This is done directly:

```
>> gbd = im2double(gb)
gbd =
     0     0
     1     1
```

If gb had been of class `uint8`, applying `im2double` to it would have resulted in an array with values

```
         0         0
    0.0039    0.0039
```

because `im2double` would have divided all the elements by 255. This did not happen in the preceding conversion because `im2double` detected that the input was a `logical` array, whose only possible values are 0 and 1. If the input in fact had been of class `uint8` and we wanted to convert it to class `double` while keeping the 0 and 1 values, we would have converted the array by writing

```
>> gbd = double(gb)
gbd =
     0     0
     1     1
```

Finally, we point out that the output of one function can be passed directly as the input to another, so we could have started with image `f` and arrived at the same result by using the one-line statement

```
>> gbd = im2double(im2bw(mat2gray(f), 0.6));
```

or by using partial groupings of these functions. Of course, the entire process could have been done in this case with a simpler command:

```
>> gbd = double(f > 2);
```

demonstrating again the compactness of the MATLAB language. ■

As the first two entries in Table 2.3 show class numeric data of class `double` requires twice as much storage as data of class `single`. In most image processing applications in which numeric processing is used, `single` precision is perfectly adequate. Therefore, unless a specific application or a MATLAB or toolbox function requires class `double`, it is good practice to work with `single` data to conserve memory. A consistent programming pattern that you will see used throughout the book to change inputs to class `single` is as follows:

Recall from Section 1.6 that we the a margin icon to denote the first use of a function developed in the book.

`tofloat`

See function `intrans` in Section 3.2.3 for an example of how `tofloat` is used.

```
[fout, revertclass] = tofloat(f);
g = some_operation(fout)
g = revertclass(g);
```

Function `tofloat` (see Appendix C for the code) converts an input image `f` to floating-point. If `f` is a `double` or `single` image, then `fout` equals `f`. Otherwise, `fout` equals `im2single(f)`. Output `revertclass` can be used to convert back to the same class as `f`. In other words, the idea is to convert the input

image to `single`, perform operations using `single` precision, and then, if so desired, convert the final output image to the same class as the input. The valid image classes for `f` are those listed in the third column of the first four entries in Table 2.4: `logical`, `uint8`, `unint16`, `int16`, `double`, and `single`.

2.8 Array Indexing

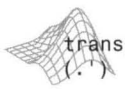
MATLAB supports a number of powerful indexing schemes that simplify array manipulation and improve the efficiency of programs. In this section we discuss and illustrate basic indexing in one and two dimensions (i.e., vectors and matrices), as well as indexing techniques useful with binary images.

2.8.1 Indexing Vectors

As discussed in Section 2.1.2, an array of dimension $1 \times N$ is called a *row vector*. The elements of such a vector can be accessed using a single index value (also called a *subscript*). Thus, `v(1)` is the first element of vector `v`, `v(2)` is its second element, and so forth. Vectors can be formed in MATLAB by enclosing the elements, separated by spaces or commas, within square brackets. For example,

```
>> v = [1 3 5 7 9]
v =
    1 3 5 7 9
>> v(2)
ans =
    3
```

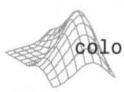
A row vector is converted to a column vector (and vice versa) using the *transpose operator* (`.'`):



Using a single quote without the period computes the conjugate transpose. When the data are real, both transposes can be used interchangeably. See Table 2.5.

```
>> w = v.'
w =
    1
    3
    5
    7
    9
```

To access *blocks* of elements, we use MATLAB's *colon* notation. For example, to access the first three elements of `v` we write

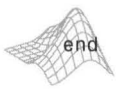



```
>> v(1:3)
ans =
    1   3   5
```

Similarly, we can access the second through the fourth elements

```
>> v(2:4)
ans =
    3    5    7
```

or all the elements from, say, the third through the last element:




```
>> v(3:end)
ans =
    5    7    9
```

where `end` signifies the last element in the vector.

Indexing is not restricted to contiguous elements. For example,

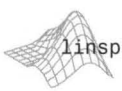
```
>> v(1:2:end)
ans =
    1    5    9
```

The notation `1:2:end` says to start at `1`, count up by `2`, and stop when the count reaches the last element. The steps can be negative:

```
>> v(end:-2:1)
ans =
    9    5    1
```

Here, the index count started at the last element, decreased by `2`, and stopped when it reached the first element.

Function `linspace`, with syntax




```
x = linspace(a, b, n)
```

generates a row vector `x` of `n` elements linearly-spaced between, and including, `a` and `b`. We use this function in several places in later chapters. A vector can even be used as an index into another vector. For example, we can select the first, fourth, and fifth elements of `v` using the command

```
>> v([1 4 5])
ans =
    1    7    9
```

As we show in the following section, the ability to use a vector as an index into another vector also plays a key role in matrix indexing.

2.8.2 Indexing Matrices

Matrices can be represented conveniently in MATLAB as a sequence of row vectors enclosed by square brackets and separated by semicolons. For example, typing

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```

gives the 3×3 matrix

```
A =
    1    2    3
    4    5    6
    7    8    9
```

Note that the use of semicolons inside square brackets is different from their use mentioned earlier to suppress output or to write multiple commands in a single line. We select elements in a matrix just as we did for vectors, but now we need two indices: one to establish a row location, and the other for the corresponding column. For example, to extract the element in the second row, third column of matrix `A`, we write

```
>> A(2, 3)
ans =
    6
```

A submatrix of `A` can be extracted by specifying a vector of values for both the row and the column indices. For example, the following statement extracts the submatrix of `A` containing rows 1 and 2 and columns 1, 2, and 3:

```
>> T2 = A([1 2], [1 2 3])
T2 =
    1    2    3
    4    5    6
```

Because the expression `1:K` creates a vector of integer values from `1` through `K`, the preceding statement could be written also as:

```
>> T2 = A(1:2, 1:3)
T2 =
    1    2    3
    4    5    6
```

The row and column indices do not have to be contiguous, nor do they have to be in ascending order. For example,

```
>> E = A([1 3], [3 2])
E =
     3     2
     9     8
```

The notation `A([a b], [c d])` selects the elements in `A` with coordinates `(a, c)`, `(a, d)`, `(b, c)`, and `(b, d)`. Thus, when we let `E = A([1 3], [3 2])`, we are selecting the following elements in `A`: `A(1, 3)`, `A(1, 2)`, `A(3, 3)`, and `A(3, 2)`.

The row or column index can also be a single colon. A colon in the row index position is shorthand notation for selecting all rows. Similarly, a colon in the column index position selects all columns. For example, the following statement selects the entire 3rd column of `A`:

```
>> C3 = A(:, 3)
C3 =
     3
     6
     9
```

Similarly, this statement extracts the second row:

```
>> R2 = A(2, :)
R2 =
     4     5     6
```

Any of the preceding forms of indexing can be used on the left-hand side of an assignment statement. The next two statements create a copy, `B`, of matrix `A`, and then assign the value `0` to all elements in the 3rd column of `B`.

```
>> B = A;
>> B(:, 3) = 0
B =
     1     2     0
     4     5     0
     7     8     0
```

The keyword `end`, when it appears in the row index position, is shorthand notation for the last row. When `end` appears in the column index position, it indicates the last column. For example, the following statement finds the element in the last row and last column of `A`:

```
>> A(end, end)
ans =
     9
```

When used for indexing, the end keyword can be mixed with arithmetic operations, as well as with the colon operator. For example:

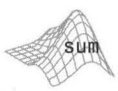
```
>> A(end, end - 2)
ans =
     7

>> A(2:end, end:-2:1)
ans =
     6     4
     9     7
```

2.8.3 Indexing with a Single Colon

The use of a single colon as an index into a matrix selects all the elements of the array and arranges them (in column order) into a single column vector. For example, with reference to matrix T2 in the previous section,

```
>> v = T2(:)
v =
     1
     4
     2
     5
     3
     6
```

This use of the colon is helpful when, for example, we want to find the sum of all the elements of a matrix. One approach is to call function sum twice:




```
>> col_sums = sum(A)
col_sums =
     111    15    112
```

Function sum computes the sum of each column of A, storing the results into a row vector. Then we call sum again, passing it the vector of column sums:

```
>> total_sum = sum(col_sums)
total_sum =
     238
```

An easier procedure is to use single-colon indexing to convert A to a column vector, and pass the result to sum:

```
>> total_sum = sum(A(:))
total_sum =
    238
```

2.8.4 Logical Indexing

Another form of indexing that you will find quite useful is *logical indexing*. A logical indexing expression has the form A(D), where A is an array and D is a logical array of the same size as A. The expression A(D) extracts all the elements of A corresponding to the 1-valued elements of D. For example,

```
>> D = logical([1 0 0; 0 0 1; 0 0 0])
D =
    1   0   0
    0   0   1
    0   0   0

>> A(D)
ans =
    1
    6
```

where A is as defined at the beginning of Section 2.8.2. The output of this method of logical indexing always is a column vector.

Logical indexing can be used also on the left-hand side of an assignment statement. For example, using the same D as above,

```
>> A(D) = [30 40]
A =
    30   2   3
     4   5  40
     7   8   9
```

In the preceding assignment, the number of elements on the right-hand side matched the number of 1-valued elements of D. Alternatively, the right-hand side can be a scalar, like this:

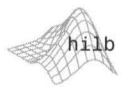
```
>> A(D) = 100
A =
```

```
100     2     3
  4     5   100
  7     8     9
```

Because binary images are represented as logical arrays, they can be used directly in logical indexing expressions to extract pixel values in an image that correspond to 1-valued pixels in a binary image. You will see numerous examples later in the book that use binary images and logical indexing.

2.8.5 Linear Indexing

The final category of indexing useful for image processing is *linear indexing*. A linear indexing expression is one that uses a *single* subscript to index a matrix or higher-dimensional array. To illustrate the concept we will use a 4×4 *Hilbert matrix* as an example:




```
>> H = hilb(4)
H =
    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429
```

`H([2 11])` is an example of a linear indexing expression:

```
>> H([2 11])
ans =
    0.5000    0.2000
```

To see how this type of indexing works, number the elements of `H` from the first to the last column in the order shown:

1.0000^{1}	0.5000^{5}	0.3333^{9}	0.2500^{13}
0.5000^{2}	0.3333^{6}	0.2500^{10}	0.2000^{14}
0.3333^{3}	0.2500^{7}	0.2000^{11}	0.1667^{15}
0.2500^{4}	0.2000^{8}	0.1667^{12}	0.1429^{16}

Here you can see that `H([2 11])` extracts the 2nd and 11th elements of `H`, based on the preceding numbering scheme.

In image processing, linear indexing is useful for extracting a set of pixel values from arbitrary locations. For example, suppose we want an expression that extracts the values of `H` at row-column coordinates (1, 3), (2, 4), and (4, 3):

```
>> r = [1 2 4];
>> c = [3 4 3];
```

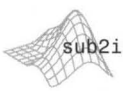
Expression `H(r, c)` does not do what we want, as you can see:

```
>> H(r, c)
ans =
    0.3333    0.2500    0.3333
    0.2500    0.2000    0.2500
    0.1667    0.1429    0.1667
```

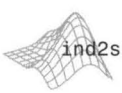
Instead, we convert the *row-column* coordinates to linear index values, as follows:

```
>> M = size(H, 1);
>> linear_indices = M*(c - 1) + r
linear_indices =
    9   14   12

>> H(linear_indices)
ans =
    0.3333    0.2000    0.1667
```

MATLAB functions `sub2ind` and `ind2sub` convert back and forth between row-column subscripts and linear indices. For example,




```
>> linear_indices = sub2ind(size(H), r, c)
linear_indices =
    9   14   12
```



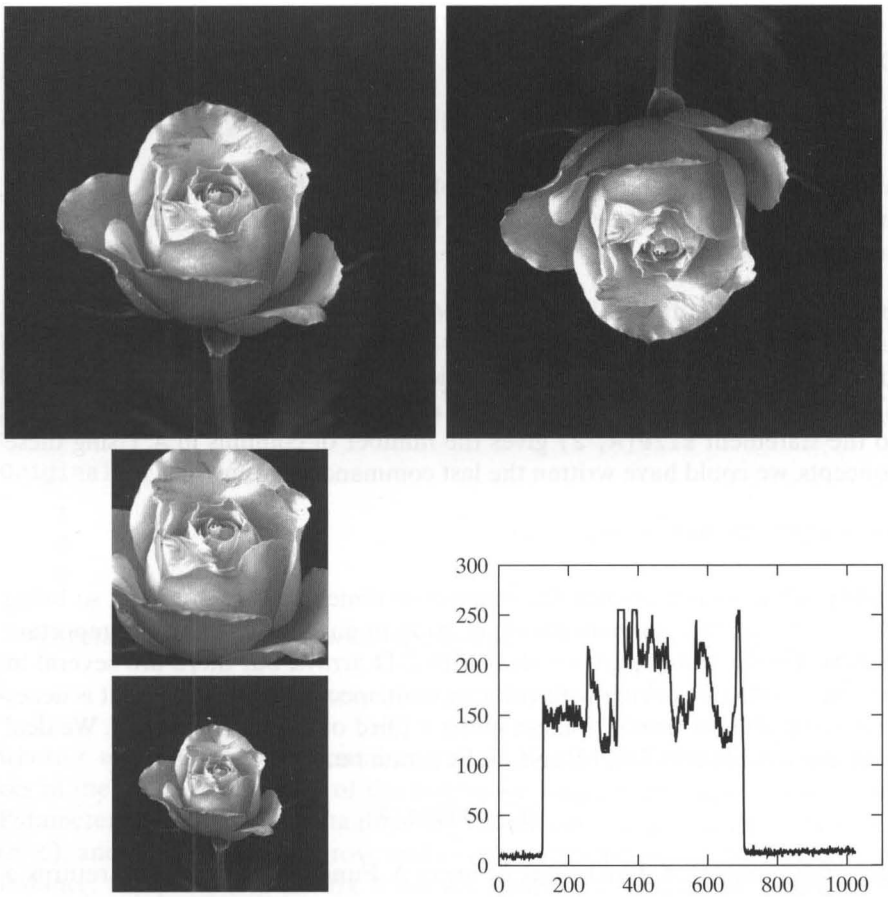

```
>> [r, c] = ind2sub(size(H), linear_indices)
r =
    1   2   4
c =
    3   4   3
```

Linear indexing is a basic staple in vectorizing loops for program optimization, as discussed in Section 2.10.5.

EXAMPLE 2.5: Some simple image operations using array indexing.

■ The image in Fig. 2.7(a) is a 1024×1024 gray-scale image, `f`, of class `uint8`. The image in Fig. 2.7(b) was flipped vertically using the statement

```
>> fp = f(end:-1:1, :);
```



a b
c
d e

FIGURE 2.7
Results obtained using array indexing. (a) Original image. (b) Image flipped vertically. (c) Cropped image. (d) Subsampled image. (e) A horizontal scan line through the middle of the image in (a).

The image in Fig. 2.7(c) is a section out of image (a), obtained using the command

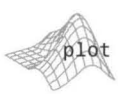
```
>> fc = f(257:768, 257:768);
```

Similarly, Fig. 2.7(d) shows a subsampled image obtained using the statement

```
>> fs = f(1:2:end, 1:2:end);
```

Finally, Fig. 2.7(e) shows a horizontal scan line through the middle of Fig. 2.7(a), obtained using the command

```
>> plot(f(512, :))
```



Function `plot` is discussed in Section 3.3.1. ■

2.8.6 Selecting Array Dimensions

Operations of the form

```
operation(A, dim)
```

where `operation` denotes an applicable MATLAB operation, `A` is an array, and `dim` is a scalar, are used frequently in this book. For example, if `A` is a 2-D array, the statement

```
>> k = size(A, 1);
```

gives the size of `A` along its first dimension (i.e., it gives the number of rows in `A`). Similarly, the second dimension of an array is in the horizontal direction, so the statement `size(A, 2)` gives the number of columns in `A`. Using these concepts, we could have written the last command in Example 2.5 as

```
>> plot(f(size(f, 1)/2, :))
```

MATLAB does not restrict the number of dimensions of an array, so being able to extract the components of an array in any dimension is an important feature. For the most part, we deal with 2-D arrays, but there are several instances (as when working with color or multispectral images) when it is necessary to be able to "stack" images along a third or higher dimension. We deal with this in Chapters 7, 8, 12, and 13. Function `ndims`, with syntax

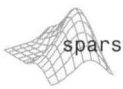



```
d = ndims(A)
```

gives the number of dimensions of array `A`. Function `ndims` never returns a value less than 2 because even scalars are considered two dimensional, in the sense that they are arrays of size 1×1.

2.8.7 Sparse Matrices

When a matrix has a large number of 0s, it is advantageous to express it in *sparse* form to reduce storage requirements. Function `sparse` converts a matrix to sparse form by "squeezing out" all zero elements. The basic syntax for this function is




```
S = sparse(A)
```

For example, if

```
>> A = [1 0 0; 0 3 4; 0 2 0]
A =
     1     0     0
     0     3     4
     0     2     0
```

Then

```
>> S = sparse(A)
S =
    (1,1)    1
    (2,2)    3
    (3,2)    2
    (2,3)    4
```

from which we see that S contains only the (row, col) locations of nonzero elements (note that the elements are sorted by columns). To recover the original (full) matrix, we use function full:




```
>> Original = full(S)
Original =
    1    0    0
    0    3    4
    0    2    0
```

A syntax used sometimes with function sparse has five inputs:

```
S = sparse(r, c, s, m, n)
```

The syntax sparse(A) requires that there be enough memory to hold the entire matrix. When that is not the case, and the location and values of all nonzero elements are known, the alternate syntax shown here provides a solution for generating a sparse matrix.

where r and c are vectors containing, respectively, the row and column indices of the nonzero elements of the matrix we wish to express in sparse form. Parameter s is a vector containing the values corresponding to index pairs (r, c), and m and n are the row and column dimensions of the matrix. For instance, the preceding matrix S can be generated directly using the command

```
>> S = sparse([1 2 3 2], [1 2 2 3], [1 3 2 4], 3, 3)
S =
    (1,1)    1
    (2,2)    3
    (3,2)    2
    (2,3)    4
```

Arithmetic and other operations (Section 2.10.2) on sparse matrices are carried out in exactly the same way as with full matrices. There are a number of other syntax forms for function sparse, as detailed in the help page for this function.

2.9 Some Important Standard Arrays

Sometimes, it is useful to be able to generate image arrays with known characteristics to try out ideas and to test the syntax of functions during development. In this section we introduce eight array-generating functions that are used in

later chapters. If only one argument is included in any of the following functions, the result is a square array.

- `zeros(M, N)` generates an `M`×`N` matrix of 0s of class `double`.
- `ones(M, N)` generates an `M`×`N` matrix of 1s of class `double`.
- `true(M, N)` generates an `M`×`N` `logical` matrix of 1s.
- `false(M, N)` generates an `M`×`N` `logical` matrix of 0s.
- `magic(M)` generates an `M`×`M` "magic square." This is a square array in which the sum along any row, column, or main diagonal, is the same. Magic squares are useful arrays for testing purposes because they are easy to generate and their numbers are integers.
- `eye(M)` generates an `M`×`M` identity matrix.
- `rand(M, N)` generates an `M`×`N` matrix whose entries are uniformly distributed random numbers in the interval [0, 1].
- `randn(M, N)` generates an `M`×`N` matrix whose numbers are normally distributed (i.e., Gaussian) random numbers with mean 0 and variance 1.

For example,

```
>> A = 5*ones(3, 3)
A =
     5     5     5
     5     5     5
     5     5     5

>> magic(3)
ans =
     8     1     6
     3     5     7
     4     9     2

>> B = rand(2, 4)
B =
    0.2311    0.4860    0.7621    0.0185
    0.6068    0.8913    0.4565    0.8214
```

2.10 Introduction to M-Function Programming

One of the most powerful features of MATLAB is the capability it provides users to program their own new functions. As you will learn shortly, MATLAB function programming is flexible and particularly easy to learn.

2.10.1 M-Files

M-files in MATLAB (see Section 1.3) can be scripts that simply execute a series of MATLAB statements, or they can be functions that can accept arguments and can produce one or more outputs. The focus of this section in on M-

file functions. These functions extend the capabilities of both MATLAB and the Image Processing Toolbox to address specific, user-defined applications.

M-files are created using a text editor and are stored with a name of the form `filename.m`, such as `average.m` and `filter.m`. The components of a function M-file are

- The function definition line
- The H1 line
- Help text
- The function body
- Comments

The *function definition line* has the form

```
function [outputs] = name(inputs)
```

For example, a function to compute the sum and product (two different outputs) of two images would have the form

```
function [s, p] = sumprod(f, g)
```

where `f` and `g` are the input images, `s` is the sum image, and `p` is the product image. The name `sumprod` is chosen arbitrarily (subject to the constraints at the end of this paragraph), but the word `function` always appears on the left, in the form shown. Note that the output arguments are enclosed by square brackets and the inputs are enclosed by parentheses. If the function has a single output argument, it is acceptable to list the argument without brackets. If the function has no output, only the word `function` is used, without brackets or equal sign. Function names must begin with a letter, and the remaining characters can be any combination of letters, numbers, and underscores. No spaces are allowed. MATLAB recognizes function names up to 63 characters long. Additional characters are ignored.

Functions can be called at the command prompt. For example,

```
>> [s, p] = sumprod(f, g);
```

or they can be used as elements of other functions, in which case they become *subfunctions*. As noted in the previous paragraph, if the output has a single argument, it is acceptable to write it without the brackets, as in

```
>> y = sum(x);
```

The *H1 line* is the first text line. It is a *single* comment line that follows the function definition line. There can be no blank lines or leading spaces between the H1 line and the function definition line. An example of an H1 line is

```
%SUMPROD Computes the sum and product of two images.
```

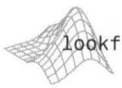
It is customary to omit the space between % and the first word in the H1 line.

The H1 line is the first text that appears when a user types




```
>> help function_name
```

at the MATLAB prompt. Typing `lookfor keyword` displays all the H1 lines containing the string `keyword`. This line provides important summary information about the M-file, so it should be as descriptive as possible.



Help text is a text block that follows the H1 line, without any blank lines in between the two. Help text is used to provide comments and on-screen help for the function. When a user types `help function_name` at the prompt, MATLAB displays all comment lines that appear between the function definition line and the first noncomment (executable or blank) line. The help system ignores any comment lines that appear after the Help text block.

The *function body* contains all the MATLAB code that performs computations and assigns values to output arguments. Several examples of MATLAB code are given later in this chapter.

All lines preceded by the symbol "%" that are not the H1 line or Help text are considered function *comment* lines and are not considered part of the Help text block. It is permissible to append comments to the end of a line of code.

M-files can be created and edited using any text editor and saved with the extension `.m` in a specified directory, typically in the MATLAB search path. Another way to create or edit an M-file is to use the `edit` function at the prompt. For example,




```
>> edit sumprod
```

opens for editing the file `sumprod.m` if the file exists in a directory that is in the MATLAB path or in the Current Directory. If the file cannot be found, MATLAB gives the user the option to create it. The MATLAB editor window has numerous pull-down menus for tasks such as saving, viewing, and debugging files. Because it performs some simple checks and uses color to differentiate between various elements of code, the MATLAB text editor is recommended as the tool of choice for writing and editing M-functions.

2.10.2 Operators

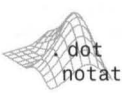
MATLAB operators are grouped into three main categories:

- Arithmetic operators that perform numeric computations
- Relational operators that compare operands quantitatively
- Logical operators that perform the functions AND, OR, and NOT

These are discussed in the remainder of this section.

Arithmetic Operators

MATLAB has two different types of arithmetic operations. *Matrix arithmetic operations* are defined by the rules of linear algebra. *Array arithmetic operations* are carried out element by element and can be used with multidimensional arrays. The period (dot) character (`.`) distinguishes array operations



from matrix operations. For example, `A*B` indicates matrix multiplication in the traditional sense, whereas `A.*B` indicates array multiplication, in the sense that the result is an array, the same size as `A` and `B`, in which each element is the product of corresponding elements of `A` and `B`. In other words, if `C = A.*B`, then `C(I, J) = A(I, J)*B(I, J)`. Because matrix and array operations are the same for addition and subtraction, the character pairs `.+` and `.–` are not used.

When writing an expression such as `B = A`, MATLAB makes a "note" that `B` is equal to `A`, but does not actually copy the data into `B` unless the contents of `A` change later in the program. This is an important point because using different variables to "store" the same information sometimes can enhance code clarity and readability. Thus, the fact that MATLAB does not duplicate information unless it is absolutely necessary is worth remembering when writing MATLAB code. Table 2.5 lists the MATLAB arithmetic operators, where `A` and `B` are matrices or arrays and `a` and `b` are scalars. All operands can be real or complex. The dot shown in the array operators is not necessary if the operands are scalars. Because images are 2-D arrays, which are equivalent to matrices, all the operators in the table are applicable to images.

The difference between *array* and *matrix operations* is important. For example, consider the following:

Throughout the book, we use the term *array operations* interchangeably with the terminology *operations between pairs of corresponding elements*, and also *elementwise operations*.

TABLE 2.5 Array and matrix arithmetic operators. Characters `a` and `b` are scalars.

Operator	Name	Comments and Examples
+	Array and matrix addition	`a + b`, `A + B`, or `a + A`.
-	Array and matrix subtraction	`a - b`, `A - B`, `A - a`, or `a - A`.
.*	Array multiplication	`Cv= A.*B`, `C(I, J) = A(I, J)*B(I, J)`.
*	Matrix multiplication	`A*B`, standard matrix multiplication, or `a*A`, multiplication of a scalar times all elements of `A`.
./	Array right division†	`C = A./B`, `C(I, J) = A(I, J)/B(I, J)`.
.\	Array left division†	`C = A.\B`, `C(I, J) = B(I, J)/A(I, J)`.
/	Matrix right division	`A/B` is the preferred way to compute `A*inv(B)`.
\	Matrix left division	`A\B` is the preferred way to compute `inv(A)*B`.
.^	Array power	If `C = A.^B`, then `C(I, J) = A(I, J)^B(I, J)`.
^	Matrix power	See `help` for a discussion of this operator.
.'	Vector and matrix transpose	`A.'`, standard vector and matrix transpose.
'	Vector and matrix complex conjugate transpose	`A'`, standard vector and matrix conjugate transpose. When `A` is real `A.' = A'`.
+	Unary plus	`+A` is the same as `0 + A`.
-	Unary minus	`-A` is the same as `0 - A` or `-1*A`.
:	Colon	Discussed in Section 2.8.1.

† In division, if the denominator is 0, MATLAB reports the result as `Inf` (denoting infinity). If both the numerator and denominator are 0, the result is reported as `NaN` (Not a Number).

$$A = \begin{bmatrix} a1 & a2 \\ a3 & a4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b1 & b2 \\ b3 & b4 \end{bmatrix}$$

The *array product* of A and B gives the result

$$A.*B = \begin{bmatrix} a1b1 & a2b2 \\ a3b3 & a4b4 \end{bmatrix}$$

whereas the *matrix product* yields the familiar result:

$$A*B = \begin{bmatrix} a1b1 + a2b3 & a1b2 + a2b4 \\ a3b1 + a4b3 & a3b2 + a4b4 \end{bmatrix}$$

Most of the arithmetic, relational, and logical operations involving images are array operations.

Example 2.6, to follow, uses functions `max` and `min`. The former function has the syntax forms



The syntax forms shown for `max` apply also to function `min`.

```
C = max(A)
C = max(A, B)
C = max(A, [ ], dim)
[C, I] = max(...)
```

In the first form, if A is a vector, `max(A)` returns its largest element; if A is a matrix, then `max(A)` treats the columns of A as vectors and returns a row vector containing the maximum element from each column. In the second form, `max(A, B)` returns an array the same size as A and B with the largest elements taken from A or B. In the third form, `max(A, [ ], dim)` returns the largest elements along the dimension of A specified by scalar `dim`. For example, `max(A, [ ], 1)` produces the maximum values along the first dimension (the rows) of A. Finally, `[C, I] = max(...)` also finds the indices of the maximum values of A, and returns them in output vector `I`. If there are duplicate maximum values, the index of the first one found is returned. The dots indicate the syntax used on the right of any of the previous three forms. Function `min` has the same syntax forms just described for `max`.

EXAMPLE 2.6: Illustration of arithmetic operators and functions `max` and `min`.

■ Suppose that we want to write an M-function, call it `imblend`, that forms a new image as an equally-weighted sum of two input images. The function should output the new image, as well as the maximum and minimum values of the new image. Using the MATLAB editor we write the desired function as follows:

```
function [w, wmax, wmin] = imblend(f, g)
%IMBLEND Weighted sum of two images.
%   [W, WMAX, WMIN] = IMBLEND(F, G) computes a weighted sum (W) of
%   two input images, F and G. IMBLEND also computes the maximum
%   (WMAX) and minimum (WMIN) values of W.  F and G must be of
%   the same size and numeric class. The output image is of the
%   same class as the input images.
```

```
w1 = 0.5 * f;
w2 = 0.5 * g;
w = w1 + w2;

wmax = max(w(:));
wmin = min(w(:));
```

Observe the use of single-colon indexing, as discussed in Section 2.8.1, to compute the minimum and maximum values. Suppose that `f = [1 2; 3 4]` and `g = [1 2; 2 1]`. Calling `imblend` with these inputs results in the following output:

```
>> [w, wmax, wmin] = imblend(f, g)
w =
    1.0000    2.0000
    2.5000    2.5000
wmax =
    2.5000
wmin =
    1
```

Note in the code for `imblend` that the input images, `f` and `g`, were multiplied by the weights (0.5) first before being added together. Instead, we could have used the statement

```
>> w = 0.5 * (f + g);
```

However, this expression does not work well for integer classes because when MATLAB evaluates the subexpression `(f + g)`, it saturates any values that overflow the range of the class of `f` and `g`. For example, consider the following scalars:

```
>> f = uint8(100);
>> g = uint8(200);
>> t = f + g
t =
    255
```

Instead of getting a sum of 300, the computed sum saturated to the maximum value for the `uint8` class. So, when we multiply the sum by 0.5, we get an incorrect result:

```
>> d = 0.5 * t
d =
    128
```

Compare this with the result when we multiply by the weights first before adding:

```
>> e1 = 0.5 * f
e1 =
    50

>> e2 = 0.5 * g
e2 =
    100

>> e = w1 + w2
e =
    150
```

A good alternative is to use the image arithmetic function `imlincomb`, which computes a weighted sum of images, for any set of weights and any number of images. The calling syntax for this function is




```
g = imlincomb(k1, f1, k2, f2,...)
```

For example, using the previous scalar values,

```
>> w = imlincomb(0.5, f, 0.5, g)
w =
    150
```

Typing `help imblend` at the command prompt results in the following output:

```
%IMBLEND Weighted sum of two images.
%   [W, WMAX, WMIN] = IMBLEND(F, G) computes a weighted sum (W) of
%   two input images, F and G.  IMBLEND also computes the maximum
%   (WMAX) and minimum (WMIN) values of W.  F and G must be of
%   the same size and numeric class.  The output image is of the
%   same class as the input images.
```

■

Relational Operators

MATLAB's relational operators are listed in Table 2.6. These are array operators; that is, they compare corresponding pairs of elements in arrays of equal dimensions.

EXAMPLE 2.7: Relational operators.

■ Although the key use of relational operators is in flow control (e.g., in `if` statements), which is discussed in Section 2.10.3, we illustrate briefly how these operators can be used directly on arrays. Consider the following:

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```

Operator	Name
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

TABLE 2.6 Relational operators.

```
A =
     1     2     3
     4     5     6
     7     8     9

>> B = [0 2 4; 3 5 6; 3 4 9]
B =
     0     2     4
     3     5     6
     3     4     9

>> A == B
ans =
     0     1     0
     0     1     1
     0     0     1
```

We see that the operation `A == B` produces a logical array of the same dimensions as `A` and `B`, with 1s in locations where the corresponding elements of `A` and `B` match, and 0s elsewhere. As another illustration, the statement

```
>> A >= B
ans =
     1     1     0
     1     1     1
     1     1     1
```

produces a logical array with 1s where the elements of `A` are greater than or equal to the corresponding elements of `B`, and 0s elsewhere. ■

In relational operators, both operands must have the same dimensions unless one operand is a scalar. In this case, MATLAB tests the scalar against every element of the other operand, yielding a logical array of the same size as the operand, with 1s in locations where the specified relation is satisfied and 0s elsewhere. If both operands are scalars, the result is a 1 if the specified relation is satisfied and 0 otherwise.

Logical Operators and Functions

Table 2.7 lists MATLAB's logical operators, and the following example illustrates some of their properties. Unlike most common interpretations of logical operators, the operators in Table 2.7 can operate on both logical *and* numeric data. MATLAB treats a logical 1 or nonzero numeric quantity as `true`, and a logical 0 or numeric 0 as `false` in all logical tests. For instance, the AND of two operands is 1 if both operands are logical 1s or both are nonzero numbers. The AND operation is 0 if either of its operands is logically or numerically 0, or if they both are logically or numerically 0.

The operators `&` and `|` operate on arrays; they compute `AND` and `OR`, respectively, on corresponding elements of their inputs. The operators `&&` and `||` operate only on scalars. They are used primarily with the various forms of `if`, and with `while` and `for` loops, all of which are discussed in Section 2.10.3.

EXAMPLE 2.8: Logical operators.

■ Consider the AND operation on the following numeric arrays:

```
>> A = [1 2 0; 0 4 5];
>> B = [1 -2 3; 0 1 1];
>> A & B

ans =

     1     1     0
     0     1     1
```

We see that the `&` operator produces a logical array that is of the same size as the input arrays and has a 1 at locations where both operands are nonzero and 0s elsewhere. Again, note that all operations are done on pairs of corresponding elements of the arrays. The `|` operator works in a similar manner. An `|` expression is `true` if either operand is a logical 1 or nonzero numerical quantity, or if they both are logical 1s or nonzero numbers; otherwise it is

TABLE 2.7 Logical operators.

Operator	Description
&	Elementwise AND
|	Elementwise OR
~	Elementwise and scalar NOT
&&	Scalar AND
||	Scalar OR

false. The ~ operator works with a single operand. Logically, if the operand is true, the ~ operator converts it to false. When using ~ with numeric data, any nonzero operand becomes 0, and any zero operand becomes 1. If you try to use the scalar logical operators && or || with nonscalar operands, MATLAB will issue an error. ■

MATLAB also supports the logical functions summarized in Table 2.8. The all and any functions are particularly useful in programming.

EXAMPLE 2.9: Logical functions.

■ Consider the arrays A = [1 2 3; 4 5 6] and B = [0 –1 1; 0 0 2]. Substituting these arrays into the functions in Table 2.8 yield the following results:

```
>> xor(A, B)
ans =
    1   0   0
    1   1   0
>> all(A)
ans =
    1   1   1
>> any(A)
ans =
    1   1   1
>> all(B)
ans =
    0   0   1
>> any(B)
ans =
    0   1   1
```

TABLE 2.8 Logical functions.

Operator	Comments
xor (exclusive OR)	The xor function returns a 1 only if both operands are logically different; otherwise xor returns a 0.
all	The all function returns a 1 if all the elements in a vector are nonzero; otherwise all returns a 0. This function operates columnwise on matrices.
any	The any function returns a 1 if any of the elements in a vector is nonzero; otherwise any returns a 0. This function operates columnwise on matrices.

Note how functions `all` and `any` operate on columns of `A` and `B`. For instance, the first two elements of the vector produced by `all(B)` are `0` because each of the first two columns of `B` contains at least one `0`; the last element is `1` because all elements in the last column of `B` are nonzero. ■

In addition to the functions listed in Table 2.8, MATLAB provides a number of other functions that test for the existence of specific conditions or values and return logical results. Some of these functions are listed in Table 2.9. A few of them deal with terms and concepts discussed earlier in this chapter; others are used in subsequent discussions. The functions in Table 2.9 return a logical 1 when the condition being tested is true; otherwise they return a logical 0. When the argument is an array, some of the functions in Table 2.9 yield an array the same size as the argument containing logical 1s in the locations that satisfy

TABLE 2.9
Some functions that return a logical 1 or a logical 0, depending on whether the value or condition in their arguments is `true` or `false`. Type `is*` in the help documentation for a complete list.

Function	Description
`iscell(C)`	True if `C` is a cell array.
`iscellstr(s)`	True if `s` is a cell array of strings.
`ischar(s)`	True if `s` is a character string.
`isempty(A)`	True if `A` is the empty array, [ ].
`isequal(A, B)`	True if `A` and `B` have identical elements and dimensions.
`isfield(S, 'name')`	True if `'name'` is a field of structure `S`.
`isfinite(A)`	True in the locations of array `A` that are finite.
`isinf(A)`	True in the locations of array `A` that are infinite.
`isinteger(A)`	True if `A` is an integer array.
`isletter(A)`	True in the locations of `A` that are letters of the alphabet.
`islogical(A)`	True if `A` is a logical array.
`ismember(A, B)`	True in locations where elements of `A` are also in `B`.
`isnan(A)`	True in the locations of `A` that are `NaN`s (see Table 2.10 for a definition of `NaN`).
`isnumeric(A)`	True if `A` is a numeric array.
`isprime(A)`	True in locations of `A` that are prime numbers.
`isreal(A)`	True if the elements of `A` have no imaginary parts.
`isscalar(A)`	True if `A` has exactly one element.
`isspace(A)`	True at locations where the elements of `A` are whitespace characters.
`issparse(A)`	True if `A` is a sparse matrix.
`isstruct(S)`	True if `S` is a structure.
`isvector(A)`	True if A is a row or column vector.

the test performed by the function, and logical 0s elsewhere. For example, if `A = [1 2; 3 1/0]`, the function `isfinite(A)` returns the matrix `[1 1; 1 0]`, where the `0` (false) entry indicates that the last element of `A` is not finite.

Some Important Values

The functions in Table 2.10 return values that are used extensively in MATLAB programming. For example, `eps` typically is added to denominators in expressions to prevent overflow when a denominator becomes zero.

Floating-Point Number Representation

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter `e` to specify a power-of-ten scale factor. Imaginary numbers use either `i` or `j` as a suffix. Some examples of valid number representations are

```
3           -99            0.0001
9.6397238    1.60210e-20   6.02252e23
1i          -3.14159j      3e5i
```

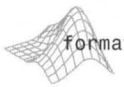
By default, numbers are stored internally using the long format specified by the Institute of Electrical and Electronics Engineers (IEEE) floating-point standard. Often, this format is called *double-precision floating point*, and corresponds to the MATLAB class `double`. As discussed in Section 2.5 (see Table 2.3), double-precision floating-point numbers have a precision of 16 significant decimal digits and a range of approximately $\pm 10^{+308}$. Single-precision floating-point numbers have a precision of 7 significant decimal digits and a range of approximately $\pm 10^{+38}$.

TABLE 2.10
Some important functions and constants.

Function	Value Returned
`ans`	Most recent answer (variable). If no output variable is assigned to an expression, MATLAB automatically stores the result in `ans`.
`eps`	Floating-point relative accuracy. This is the distance between 1.0 and the next largest number representable using double-precision floating point.
`i` (or `j`)	Imaginary unit, as in `1 + 2i`.
`NaN` or `nan`	Stands for Not-a-Number (e.g., `0/0`).
`pi`	3.14159265358979
`realmax`	The largest floating-point number that your computer can represent.
`realmin`	The smallest positive floating-point number that your computer can represent.
`computer`	Your computer type.
`version`	Version number for MATLAB.
`ver`	Version information for all installed MATLAB products.

Formats

The `format` function, with the following forms




```
format
format type
format('type')
```

is used to control how numerical data is displayed in the Command Window (only the display is affected, not how MATLAB computes and stores numerical data). The first form changes the output format to the default appropriate for the class of data being used; the second changes the format to the specified *type*; and the third form is the function form of the syntax. Table 2.11 shows the format types of interest in this book, and the following examples illustrate their use by displaying `pi` in various formats.

To determine the format currently in use, we write



The syntax `get(0, 'Format')` returns the `type` of `Format` currently in use (see Table 2.12). Also, see Section 7.4 for another syntax form of function `get`.

```
>> get(0,'Format')
ans =
    short
```

When the format is set to `short`, both `pi` and `single(pi)` display as 5-digit values:

```
>> pi
ans =
    3.1416

>> single(pi)
ans =
    3.1416
```

If we set the format to `long`, then

```
>> format long
pi
ans =
    3.14159265358979

>> single(pi)
ans =
    3.1415927
```

To use exponential notation we type

Type	Result
`short`	Scaled fixed point format, with 4 digits after the decimal point. For example, `3.1416`.
`long`	Scaled fixed point format with 14 to 15 digits after the decimal point for `double`, and 7 digits after the decimal point for `single`. For example, `3.141592653589793`.
`short e`	Floating point format, with 4 digits after the decimal point. For example, `3.1416e+000`.
`long e`	Floating point format, with 14 to 15 digits after the decimal point for `double`, and 7 digits after the decimal point for `single`. For example, `3.141592653589793e+000`.
`short g`	Best (in terms of shorter output) of fixed or floating point, with 4 digits after the decimal point. For example, `3.1416`.
`long g`	Best (in terms of shorter output) of fixed or floating point, with 14 to 15 digits after the decimal point for `double`, and 7 digits after the decimal point for `single`. For example, `3.14159265358979`.
`short eng`	Engineering format that has 4 digits after the decimal point, and a power that is a multiple of three. For example, `3.1416e+000`.
`long eng`	Engineering format that has exactly 16 significant digits and a power that is a multiple of three. For example, `3.14159265358979e+000`.

TABLE 2.11 Format types. The examples are based on constant `pi`.

```
>> format short e
>> pi
ans =
    3.1416e+000
```

or, we could have used the function form of the syntax:

```
>> format('short', 'e')
```

and the result would have been the same. As an exercise, you should look up the help page for function `format` and experiment with the other format types.

2.10.3 Flow Control

The ability to control the flow of operations based on a set of predefined conditions is at the heart of all programming languages. In fact, conditional branching was one of two key developments that led to the formulation of general-purpose computers in the 1940s (the other development was the use of memory to hold stored programs and data). MATLAB provides the eight flow control statements summarized in Table 2.12. Keep in mind the observation made in the previous section that MATLAB treats a logical 1 or nonzero number as `true`, and a logical or numeric 0 as `false`.

TABLE 2.12
Flow control statements.

Statement	Description
`if`	`if`, together with `else` and `elseif`, executes a group of statements based on a specified logical condition.
`for`	Executes a group of statements a fixed (specified) number of times.
`while`	Executes a group of statements an indefinite number of times, based on a specified logical condition.
`break`	Terminates execution of a `for` or `while` loop.
`continue`	Passes control to the next iteration of a `for` or `while` loop, skipping any remaining statements in the body of the loop.
`switch`	`switch`, together with `case` and `otherwise`, executes different groups of statements, depending on a specified value or string.
`return`	Causes execution to return to the invoking function.
`try...catch`	Changes flow control if an error is detected during execution.

`if`, `else`, and `elseif`

Conditional statement `if` has the syntax

```
if expression
   statements
end
```

As discussed in connection with Table 2.7, logical AND and OR operators appearing inside `expression` should be the scalar logical operators `&&` and `||`.

The *`expression`* is evaluated and, if the evaluation yields `true`, MATLAB executes one or more commands, denoted here as *`statements`*, between the `if` and `end` lines. If *`expression`* is `false`, MATLAB skips all the statements between the `if` and `end` lines and resumes execution at the line following the `end` line. When nesting `if`s, each `if` must be paired with a matching `end`.

The `else` and `elseif` statements further conditionalize the `if` statement. The general syntax is

```
if expression1
   statements1
elseif expression2
   statements2
else
   statements3
end
```

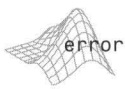
If *`expression1`* is `true`, *`statements1`* are executed and control is transferred to the `end` statement. If *`expression1`* evaluates to `false`, then *`expression2`* is evaluated. If this expression evaluates to `true`, then *`statements2`* are executed and control is transferred to the `end` statement. Otherwise (`else`) *`statements3`* are executed. Note that the `else` statement has no condition.

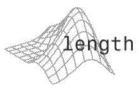
The `else` and `elseif` statements can appear by themselves after an `if` statement; they do not need to appear in pairs, as shown in the preceding general syntax. It is acceptable to have multiple `elseif` statements.

EXAMPLE 2.10: Conditional branching.

■ Suppose that we want to write a function that computes the average intensity of an image. As explained in Section 2.8.3, a two-dimensional array `f` can be converted to a column vector, `v`, by letting `v = f(:)`. Therefore, we want our function to be able to work with both vector and image inputs. The program should produce an error if the input is not a one- or two-dimensional array.

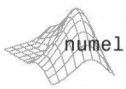
```
function av = average(A)
%AVERAGE Computes the average value of an array.
%   AV = AVERAGE(A) computes the average value of input A,
%   which must be a 1-D or 2-D array.

%   Check the validity of the input.
if ndims(A) > 2
   error('The dimensions of the input cannot exceed 2.')
end

% Compute the average
av = sum(A(:))/length(A(:));
```





Note that the input is converted to a 1-D array by using `A(:)`. In general, `length(A)` returns the size of the longest dimension of an array, `A`. In this example, because `A(:)` is a vector, `length(A)` gives the number of elements of `A`. This eliminates the need for a separate test to determine whether the input is a vector or a 2-D array. Another way to obtain the number of elements in an array directly is to use function `numel`, whose syntax is

```
n = numel(A)
```



Thus, if `A` is an image, `numel(A)` gives its number of pixels. Using this function, the last line of the previous program becomes

```
av = sum(A(:))/numel(A);
```

Finally, note that the `error` function terminates execution of the program and outputs the message contained within the parentheses (the quotes shown are required). ■

for

A `for` loop executes a group of statements a specified number of times. The syntax is

```
for index = start:increment:end
   statements
end
```

It is possible to nest two or more `for` loops, as follows:

```
for index1 = start1:increment1:end
   statements1
   for index2 = start2:increment2:end
      statements2
   end
   additional loop1 statements
end
```

For example, the following loop executes 11 times:

```
count = 0;
for k = 0:2:20
   count = count + 1;
end
```

If the loop increment is omitted, it is taken to be 1. Loop increments also can be negative, as in `k = 0:-1:-10`. Note that no semicolon is necessary at the end of a `for` line. MATLAB automatically suppresses printing the values of a loop index. As discussed in detail in Section 2.10.5, improvements in program execution speed sometimes can be achieved by replacing `for` loops with so-called *vectorized code* whenever possible.

EXAMPLE 2.11: Using a `for` loop to write multiple images to file.

■ Example 2.2 compared several images using different JPEG quality values. Here, we show how to write those files to disk using a `for` loop. Suppose that we have an image, `f`, and we want to write it to a series of JPEG files with quality factors ranging from 0 to 100 in increments of 5. Further, suppose that we want to write the JPEG files with filenames of the form `series_xxx.jpg`, where `xxx` is the quality factor. We can accomplish this using the following `for` loop:

```
for q = 0:5:100
   filename = sprintf('series_%3d.jpg', q);
   imwrite(f, filename, 'quality', q);
end
```

Function `sprintf`, whose syntax in this case is



See the help page for `sprintf` for other useful syntax forms.

```
s = sprintf('characters1%ndcharacters2', q)
```

writes formatted data as a string, `s`. In this syntax form, `characters1` and `characters2` are character strings, and `%nd` denotes a decimal number (specified by `q`) with `n` digits. In this example, `characters1` is `series_`, the value of `n` is 3, `characters2` is `.jpg`, and `q` has the values specified in the loop. ■

`while`

A `while` loop executes a group of statements for as long as the expression controlling the loop is `true`. The syntax is

```
while expression
   statements
end
```

As with the `if` statement, logical AND and OR operators appearing inside *`expression`* should be the scalar logical operators `&&` and `||`. As in the case of `for`, `while` loops can be nested:

```
while expression1
   statements1
   while expression2
      statements2
   end
   additional loop1 statements
end
```

For example, the following nested `while` loops terminate when both `a` and `b` have been reduced to 0:

```
a = 10;
b = 5;
while a
   a = a - 1;
   while b
      b = b - 1;
   end
end
```

Note that to control the loops we used MATLAB's convention of treating a numerical value in a logical context as `true` when it is nonzero and as `false` when it is 0. In other words, `while a` and `while b` evaluate to `true` as long as `a` and `b` are nonzero. As in the case of `for` loops, gains in program execution speed sometimes can be achieved by replacing `while` loops with vectorized code (Section 2.10.5).

`break`

As its name implies, `break` terminates the execution of a `for` or `while` loop. When a `break` statement is encountered, execution continues with the next statement outside the loop. In nested loops, `break` exits only from the innermost loop that contains it.

`continue`

The `continue` statement passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, `continue` passes control to the next iteration of the innermost loop enclosing it.

switch

This is the statement of choice for controlling the flow of an M-function based on different types of inputs. The syntax is

```
switch switch_expression
   case case_expression
      statement(s)
   case {case_expression1, case_expression2,...}
      statement(s)
   otherwise
      statement(s)
end
```

The `switch` construct executes groups of statements based on the value of a variable or expression. The keywords `case` and `otherwise` delineate the groups. Only the first matching `case` is executed.† There must always be an `end` to match the `switch` statement. The curly braces are used when multiple expressions are included in the same `case` statement. As an example, suppose that an M-function accepts an image `f` and converts it to a specified class, call it `newclass`. Only three image classes are acceptable for the conversion: `uint8`, `uint16`, and `double`. The following code fragment performs the desired conversion and outputs an error if the class of the input image is not one of the acceptable classes:

```
switch newclass
   case 'uint8'
      g = im2uint8(f);
   case 'uint16'
      g = im2uint16(f);
   case 'double'
      g = im2double(f);
   otherwise
      error('Unknown or improper image class.')
end
```

The `switch` construct is used extensively throughout the book.

EXAMPLE 2.12: Extracting a subimage from a given image.

■ In this example we write an M-function (based on `for` loops) to extract a rectangular subimage from an image. Although we could do the extraction using a single MATLAB statement (do it as an exercise after you read about vectorized code in Section 2.10.5), the objective here is to illustrate `for` loops. The inputs to the function are an image, the size (number of rows and columns) of the subimage we want to extract, and the coordinates of the top, left corner of the subimage. Keep in mind that the image origin in MATLAB

†Unlike the C language switch construct, MATLAB's `switch` does not "fall through." That is, `switch` executes only the first matching case; subsequent matching cases do not execute. Therefore, `break` statements are not used.

is at (1, 1), as discussed in Section 2.1.1.

```
function s = subim(f, m, n, rx, cy)
%SUBIM Extracts a subimage, s, from a given image, f.
%   The subimage is of size m-by-n, and the coordinates of its top,
%   left corner are (rx, cy).

s = zeros(m, n);
for r = 1:m
   for c = 1:n
      s(r,c) = f(r + rx - 1, c + cy - 1);
   end
end
```

As an exercise, you should implement the preceding program using `while`, instead of `for`, loops. ■

2.10.4 Function Handles

A *function handle* is a MATLAB data type that contains information used in referencing a function. One of the principal advantages of using function handles is that you can pass a function handle as an argument in a call to another function. As you will see in the next section, the fact that a function handle carries all the information needed for MATLAB to evaluate the function can lead to simpler program implementation. Function handles also can improve performance in repeated operations, and, in addition to being passed to other functions, they can be saved in data structures or files for later use.



There are two different types of function handles, both of which are created using the function handle operator, `@`. The first function handle type is the *named* (also called *simple*) *function handle*. To create a named function handle, follow the `@` operator with the name of the desired function. For example:

```
>> f = @sin
f =
    @sin
```

Function `sin` can be called indirectly by calling the function handle, `f`:

```
>> f(pi/4)
ans =
    0.7071

>> sin(pi/4)
ans =
    0.7071
```

The second function handle type is the *anonymous function handle*, which is formed from a MATLAB expression instead of a function name. The general format for constructing an anonymous function is:

```
@(input-argument-list) expression
```

For example, the following anonymous function handle squares its input:

```
>> g = @(x) x.^2;
```

and the following handle computes the square root of the sum of two squared variables:

```
>> r = @(x, y) sqrt(x.^2 + y.^2);
```

Anonymous function handles can be called just like named function handles:

```
>> g(3)
ans =
    9

>> r(3,  4)
ans =
    5
```

Many MATLAB and Image Processing Toolbox functions take function handles as input arguments. For instance, the `quad` function performs numerical integration. The function to be integrated is specified by passing a function handle as an input argument to `quad`. For example, the following statement computes the definite integral of the `sin` function over the interval `[0, pi/4]` (recall from the discussion above that `f = @sin`):



Function `quad` performs numerical integration using an adaptive Simpson quadrature approach.

```
>> quad(f, 0, pi/4)
ans =
    0.2929
```

where `f` is as defined above. Anonymous function handles can be passed to other functions in exactly the same manner. The following statement computes the definite integral of x^2 over the interval $[0, 1]$:

```
>> quad(g, 0, 1)
ans =
    0.3333
```

where `g` is as defined above. We give additional examples of function handles in the following section and in later chapters.

2.10.5 Code Optimization

As discussed in some detail in Section 1.3, MATLAB is a programming language designed specifically for array operations. Taking advantage of this fact whenever possible can result in significant increases in computational speed. In this section we discuss two important approaches for MATLAB code optimization: preallocating arrays and vectorizing loops.

Preallocating Arrays

Preallocation refers to initializing arrays before entering a `for` loop that computes the elements of the array. To illustrate why preallocation can be important, we start with a simple experiment. Suppose that we want to create a MATLAB function that computes

$$f(x) = \sin(x/100\pi)$$

for $x = 0, 1, 2, \ldots, M - 1$. Here is our first version of the function:

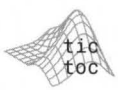
```
function y = sinfun1(M)
x = 0:M - 1;
for k = 1:numel(x)
   y(k) = sin(x(k) / (100*pi));
end
```

The output for `M = 5` is

```
>> sinfun1(5)
ans =
    0    0.0032    0.0064    0.0095    0.0127
```

MATLAB functions `tic` and `toc` can be used to measure how long a function takes to execute. We call `tic`, then call the function, and then call `toc`:

```
>> tic; sinfun1(100); toc
Elapsed time is 0.001205 seconds.
```



(If you type the preceding three statements in separate lines, the time measured will include the time required for you to type the second two lines.)

Timing functions using calls as in the preceding paragraph can produce large variations in the measured time, especially when done at the command prompt. For example, repeating the previous call gives a different result:

```
>> tic; sinfun1(100); toc
Elapsed time is 0.001197 seconds.
```

Function `timeit` can be used to obtain reliable, repeatable time measurements of function calls. The calling syntax for `timeit`[†] is

`timeit`

```
s = timeit(f)
```

where `f` is a function handle for the function to be timed, and `s` is the measured time, in seconds, required to call `f`. The function handle `f` is called with no input arguments. We can use `timeit` as follows to time `sinfun1` for `M = 100`:

```
>> M = 100;
>> f = @() sinfun1(M);
>> timeit(f)
ans =
    8.2718e-005
```

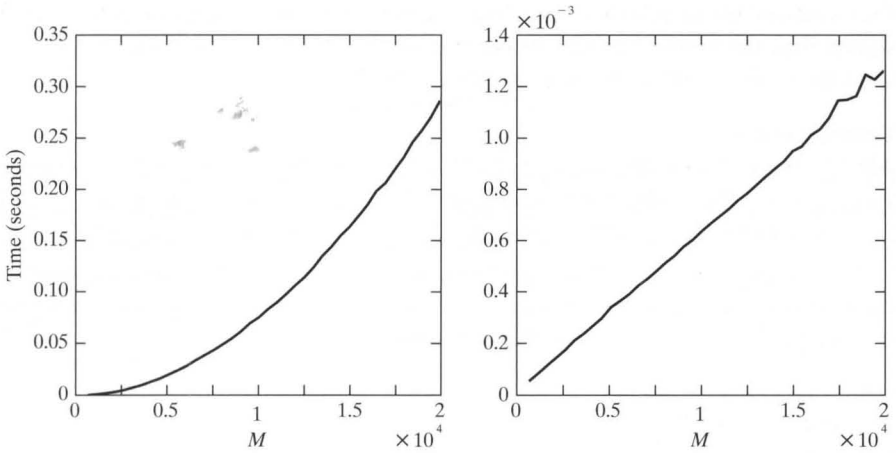
This call to function `timeit` is an excellent illustration of the power of the concept of function handles introduced in the previous section. Because it accepts a function handle with no inputs, function `timeit` is *independent* of the parameters of the function we wish to time. Instead, we delegate that task to the creation of the function handle itself. In this case, only one parameter, `M`, was necessary. But you can imagine more complex functions with numerous parameters. Because a function handle stores all the information needed to evaluate the function for which it is defined, it is possible for `timeit` to require a single input, and yet be capable of timing any function, independently of its complexity or number of parameters. This is a very useful programming feature.

Continuing with our experiment, we use `timeit` to measure how long `sinfun1` takes for `M = 500, 1000, 1500, ..., 20000`:

```
M = 500:500:20000;
for k = 1:numel(M)
   f = @() sinfun1(M(k));
   t(k) = timeit(f);
end
```

Although we might expect the time required to compute `sinfun1(M)` to be proportional to `M`, Fig. 2.8(a) shows that the time required actually grows as a function of `M^2` instead. The reason is that in `sinfun1.m` the output variable `y` grows in size by one element each time through the loop. MATLAB can handle this implicit array growth automatically, but it has to reallocate new memory space and copy the previous array elements every time the array grows. This frequent memory reallocation and copying is expensive, requiring much more time than the `sin` computation itself.

[†]It is not practical to provide a listing of function `timeit` in the book because this function contains hundreds of tedious, repeated lines of code designed to accurately determine time-measurement overhead. You can obtain a listing from: http://www.mathworks.com/matlabcentral/fileexchange/18798.



a b

FIGURE 2.8
(a) Approximate execution times for function `sinfun1` as a function of `M`. (b) Approximate times for function `sinfun2`. The glitches were caused by interval variations in memory paging. The time scales in (a) and (b) are different.

The solution to this performance problem is suggested by the MATLAB Editor, which reports for `sinfun1.m` that:

```
'y' might be growing inside a loop. Consider preallocating for speed.
```

As mentioned in Section 1.7.1, the MATLAB editor analyzes code and makes improvement suggestions. In the case of `sinfun1`, the `y` inside the `for` loop would be shown underlined in red. Putting the cursor over `y` would display the message shown here.

Preallocating `y` means initializing it to the expected output size before beginning the loop. Usually, preallocation is done using a call to function `zeros` (see Section 2.9). Our second version of the function, `sinfun2.m`, uses preallocation:

```
function y = sinfun2(M)
x = 0:M-1;
y = zeros(1, numel(x));
for k = 1:numel(x)
   y(k) = sin(x(k) / (100*pi));
end
```

Compare the time required for `sinfun1(20000)` and `sinfun2(20000)`:

```
>> timeit(@() sinfun1(20000))
ans =
    0.2852

>> timeit(@() sinfun2(20000))
ans =
    0.0013
```

Execution times depend on the machine used. The important quantity here is the *ratio* of the execution times.

The version using preallocation runs about 220 times faster. Figure 2.8(b) shows that the time required to run `sinfun2` is proportional to `M`. [Note that the time scale is different for Figs. 2.8(a) and (b).]

Vectorizing Loops

Vectorization in MATLAB refers to techniques for eliminating loops altogether, using a combination of matrix/vector operators, indexing techniques, and existing MATLAB or toolbox functions. As an example, we revisit the `sinfun` functions discussed in the previous section. Our third version of `sinfun` exploits the fact that `sin` can operate elementwise on an array input, not just on a scalar input. Function `sinfun3` has no `for` loops:

```
function y = sinfun3(M)
x = 0:M-1;
y = sin(x ./ (100*pi));
```

In older versions of MATLAB, eliminating loops by using matrix and vector operators almost always resulted in significant increases in speed. However, recent versions of MATLAB can compile simple `for` loops automatically, such as the one in `sinfun2`, to fast machine code. As a result, many `for` loops that were slow in older versions of MATLAB are no longer slower than the vectorized versions. We can see here, in fact, that `sinfun3`, with no loops, runs at about the same speed as `sinfun2`, which has a loop:

```
>> timeit(@() sinfun2(20000))
ans =
    0.0013

>> timeit(@() sinfun3(20000))
ans =
    0.0018
```

As the following example shows, gains in speed still are possible using vectorization, but the gains are not as dramatic as they used to be in earlier versions of MATLAB.

EXAMPLE 2.13: An illustration of vectorization, and introduction of function `meshgrid`.

■ In this example, we write two versions of a MATLAB function that creates a synthetic image based on the equation:

$$f(x, y) = A\sin(u_0 x + v_0 y)$$

The first function, `twodsin1`, uses two nested `for` loops to compute `f`:

```
function f = twodsin1(A, u0, v0, M, N)
f = zeros(M, N);
for c = 1:N
   v0y = v0 * (c - 1);
   for r = 1:M
```

```
        u0x = u0 * (r - 1);
        f(r, c) = A*sin(u0x + v0y);
    end
end
```

Observe the preallocation step, `f = zeros(M, N)`, before the `for` loops. We use `timeit` to see how long this function takes to create a sinusoidal image of size 512×512 pixels:

```
>> timeit(@() twodsin1(1, 1/(4*pi), 1/(4*pi), 512, 512))
ans =
    0.0471
```

Without preallocation, this function would run approximately 42 times slower, taking 1.9826 s to execute with the same input parameters.

We can display the resulting image using the auto-range syntax (`[ ]`) of `imshow`:

```
>> f = twodsin1(1, 1/(4*pi), 1/(4*pi), 512, 512);
>> imshow(f, [ ])
```

Figure 2.9 shows the result.

In our second version of the function, we vectorize it (that is, we rewrite it without using `for` loops) by using a very useful MATLAB function called `meshgrid`, with syntax

```
[C, R] = meshgrid(c, r)
```



As detailed in help, `meshgrid` has a 3-D formulation useful for evaluating functions of three variables and for constructing volumetric plots.

The input arguments `c` and `r` are vectors of horizontal (column) and vertical (row) coordinates, respectively (note that columns are listed *first*). Function `meshgrid` transforms the coordinate vectors into two arrays `C` and `R` that can be used to compute a function of two variables. For example, the following

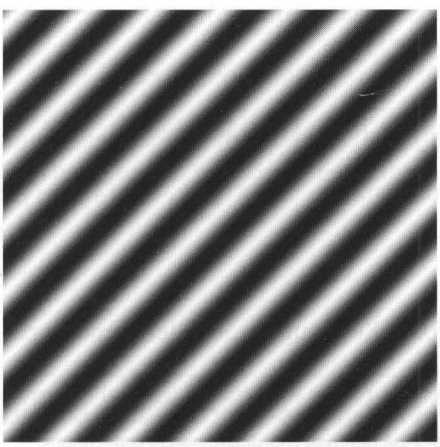


FIGURE 2.9 Sinusoidal image generated in Example 2.13.

commands use `meshgrid` to evaluate the function $z = x + y$ for integer values of x ranging from 1 to 3, and for integer values of y ranging from 10 to 14:[†]

```
>> [X, Y] = meshgrid(1:3, 10:14)
X =
     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3
Y =
    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14

>> Z = X + Y
Z =
    11    12    13
    12    13    14
    13    14    15
    14    15    16
    15    16    17
```

Finally, we use `meshgrid` to rewrite the 2-D sine function without loops:

```
function f = twodsin2(A, u0, v0, M, N)
r = 0:M - 1; % Row coordinates.
c = 0:N - 1; % Column coordinates.
[C, R] = meshgrid(c, r);
f = A*sin(u0*R + v0*C);
```

As before, we use `timeit` to measure its speed:

```
>> timeit(@() twodsin2(1, 1/(4*pi), 1/(4*pi), 512, 512))
ans =
    0.0126
```

The vectorized version takes roughly 50% less time to run. ■

[†]Function `meshgrid` assumes that x and y are horizontal (column) and vertical (row) coordinates, respectively. This is an example of the comments in Section 2.1.1 regarding the fact that MATLAB and the Image Processing Toolbox sometimes use different coordinate system conventions.

Because each new release of MATLAB tends to have improved ability to run loops faster, it is difficult to give general guidelines about when to vectorize MATLAB code. For many mathematically trained users who are familiar with matrix and vector notation, vectorized code is often more readable (it looks more "mathematical") than code based on loops. For example, compare this line from function `twodsin2`:

```
f = A*sin(u0*R + v0*C);
```

with these lines from `twodsin1` for performing the same operation:

```
for c = 1:N
   v0y = v0*(c - 1);
   for r = 1:M
      u0x = u0 * (r - 1);
      f(r, c) = A*sin(u0x + v0y);
    end
end
```

Clearly the first formulation is more concise, but the mechanics of what actually is taking place are clearer in the second.

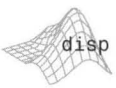
One should strive first to write code that is correct and understandable. Then, if the code does not run fast enough, use the MATLAB Profiler (see Section 1.7.1) to identify possible performance trouble spots. If any of these trouble spots are `for` loops, make sure that there are no preallocation issues and then consider using vectorization techniques. The MATLAB documentation contains further guidance about performance; search the documentation for the section titled "Techniques for Improving Performance."

2.10.6 Interactive I/O

In this section we establish a foundation for writing interactive M-functions that display information and instructions to users and accept inputs from a keyboard.

Function `disp` is used to display information on the screen. Its syntax is

```
disp(argument)
```



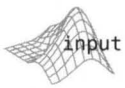
If `argument` is an array, `disp` displays its contents. If `argument` is a text string, then `disp` displays the characters in the string. For example,

```
>> A = [1 2; 3 4];
>> disp(A)
     1     2
     3     4

>> sc = 'Digital Image Processing.';
>> disp(sc)
```

```
Digital Image Processing.
>> disp('This is another way to display text.')
This is another way to display text.
```

Note that only the contents of `argument` are displayed, without words such as `ans =`, which we are accustomed to seeing on the screen when the value of a variable is displayed by omitting a semicolon at the end of a command line.

Function `input` is used for inputting data into an M-function. The basic syntax is




```
t = input('message')
```

This function outputs the words contained in `message` and waits for an input from the user, followed by a **Return** (**Enter**), and stores the input in `t`. The input can be a single number, a character string (enclosed by single quotes), a vector (enclosed by square brackets and elements separated by spaces or commas), a matrix (enclosed by square brackets and rows separated by semicolons), or any other valid MATLAB data structure. For example,

```
>> t = input('Enter your data:')
Enter your data: 25
t =
    25
>> class(t)
ans =
    double
>> t = input('Enter your data:')
Enter your data: 'abc'
t =
    abc
>> class(t)
ans =
    char
>> t = input('Enter your data:')
Enter your data: [0 1 2 3]
t =
    0   1   2   3
>> size(t)
ans =
    1   4
```

If the entries are a mixture of characters and numbers, then we use one of MATLAB's string processing functions. Of particular interest in the present discussion is function `strread`, which has the syntax

```
[a, b, c, ...] = strread(cstr, 'format', 'param', 'value')
```



See the help page for `strread` for a list of the numerous syntax forms applicable to this function.

This function reads data from the character string `cstr`, using a specified `format` and `param/value` combinations. In this chapter the formats of interest are `%f` and `%q`, to denote floating-point numbers and character strings, respectively. For `param` we use `delimiter` to denote that the entities identified in `format` will be delimited by a character specified in `value` (typically a comma or space). For example, suppose that we have the string

```
>> t = '12.6, x2y, z';
```

To read the elements of this input into three variables `a`, `b`, and `c`, we write

```
>> [a, b, c] = strread(t, '%f%q%q', 'delimiter', ',')
a =
    12.6000
b =
    'x2y'
c =
    'z'
```

Output `a` is of class `double`. The quotes around outputs `x2y` and `z` indicate that `b` and `c` are `cell` arrays, which are discussed in the next section. We convert them to character arrays simply by letting

```
>> d = char(b)
d =
    x2y
```



and similarly for `c`. The number (and order) of elements in the format string must match the number and type of expected output variables on the left. In this case we expect three inputs: one floating-point number followed by two character strings.

Function `strcmp` is used to compare strings. For example, suppose that we wish to write an M-function, `g = imnorm(f, param)`, that accepts an image, `f`, and a parameter `param` than can have one of two forms: `'norm1'` and `'norm255'`. In the first instance, `f` is to be scaled to the range [0, 1]; in the second, it is to be scaled to the range [0, 255]. The output should be of class `double` in both cases. The following code fragment accomplishes the required normalization:



Function `strcmp` compares two stringsand returns a logical `true` (`1`) if the strings are equal or a logical `false` (`0`) if they are not.

```
f = mat2gray(f);
if strcmp(param, 'norm1')
   g = f;
elseif strcmp(param, 'norm255')
    g = 255*f;
else
   error('Unknown value of param.')
end
```

An error would occur if the value specified in `param` is not `'norm1'` or `'norm255'`. Also, an error would be issued if other than all lowercase characters are used for either normalization factor. We can modify the function to accept either lower or uppercase characters by using function `strcmpi`, which performs case-insensitive string comparisons.



2.10.7 An Introduction to Cell Arrays and Structures

We conclude this chapter with a discussion of cell arrays and structures. As you will see in subsequent chapters, are used extensively in M-function programming.

Cell arrays

Cell arrays provide a way to combine a mixed set of objects (e.g., numbers, characters, matrices, other cell arrays) under one variable name. For example, suppose that we are working with (1) an `uint8` image, `f`, of size 512×512 pixels; (2) a sequence of 2-D coordinates in the form of rows of a 188×2 array, `b`; and (3) a cell array containing two character names, `char_array = {'area', 'centroid'}` (curly braces are used to enclose the contents of a cell array). These three dissimilar entities can be organized into a single variable, `C`, using cell arrays:

```
C = {f, b, char_array}
```

Typing `C` at the prompt would output the following results:

```
>> C
C =
    [512x512 uint8]    [188x2 double]    {1x2 cell}
```

In other words, the outputs shown are not the values of the various variables, but a description of some of their properties instead. To see the complete contents of an element of the cell, we enclose the numerical location of that element in curly braces. For instance, to see the contents of `char_array` we type

```
>> C{3}
ans =
    'area'  'centroid'
```

or we can use function `celldisp`:




```
>> celldisp(C{3})
ans{1} =
    area
ans{2} =
    centroid
```

Using parentheses instead of curly braces on an element of `C` gives a description of the variable:

```
>> C(3)
ans =
    {1x2 cell}
```

We can work with specified contents of a cell array by transferring them to a numeric or other pertinent form of array. For instance, to extract `f` from `C` we use

```
>> f = C{1};
```

Function `size` gives the size of a cell array:

```
>> size(C)
ans =
    1    3
```

Function `cellfun`, with syntax

```
D = cellfun('fname', C)
```



applies the function `fname` to the elements of cell array `C` and returns the results in the `double` array `D`. Each element of `D` contains the value returned by `fname` for the corresponding element in `C`. The output array `D` is the same size as the cell array `C`. For example,

```
>> D = cellfun('length', C)
D =
    512    188    2
```

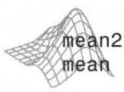
In other words, `length(f) = 512`, `length(b) = 188` and `length(char_array) = 2`. Recall from Section 2.10.3 that `length(A)` gives the size of the longest dimension of a multidimensional array `A`.

Finally, we point out that cell arrays contain *copies* of the arguments, not pointers to those arguments. Thus, if any of the arguments of `C` in the preceding example were to change after `C` was created, that change would not be reflected in `C`.

EXAMPLE 2.14: Using cell arrays.

■ Suppose that we want to write a function that outputs the average intensity of an image, its dimensions, the average intensity of its rows, and the average intensity of its columns. We can do it in the "standard" way by writing a function of the form

```
function [AI, dm, AIrows, AIcols] = image_stats(f)
dm = size(f);
AI = mean2(f);
AIrows = mean(f, 2);
AIcols = mean(f, 1);
```



`mean2(A)` computes the mean (average) value of the elements of the 2-D array `A`.

If `v` is a vector, `mean(v)` returns the mean value of the elements of `v`. If `A` is a matrix, `mean(A)` treats the columns of `A` as vectors, returning a row vector of mean values. If `A` is a multidimensional array, `mean(A, dim)` returns the mean value of the elements along the dimension specified by scalar `dim`.

where `f` is the input image and the output variables correspond to the quantities just mentioned. Using cells arrays, we would write

```
function G = image_stats(f)
G{1} = size(f);
G{2} = mean2(f);
G{3} = mean(f, 2);
G{4} = mean(f, 1);
```

Writing `G(1) = {size(f)}`, and similarly for the other terms, also is acceptable. Cell arrays can be multidimensional. For instance, the previous function could be written also as

```
function H = image_stats2(f)
H(1, 1) = {size(f)};
H(1, 2) = {mean2(f)};
H(2, 1) = {mean(f, 2)};
H(2, 2) = {mean(f, 1)};
```

Or, we could have used `H{1,1} = size(f)`, and so on for the other variables. Additional dimensions are handled in a similar manner.

Suppose that `f` is of size 512×512. Typing `G` and `H` at the prompt would give

```
>> G = image_stats(f);
>> G
G =
    [1x2 double]    [1]    [512x1 double]    [1x512 double]

>> H = image_stats2(f);
>> H
```

```
H =
    [  1x2 double]    [           1]
    [512x1 double]    [1x512 double]
```

If we want to work with any of the variables contained in `G`, we extract it by addressing a specific element of the cell array, as before. For instance, if we want to work with the size of `f`, we write

```
>> v = G{1}
```

or

```
>> v = H{1,1}
```

where `v` is a 1×2 vector. Note that we did not use the familiar command `[M, N] = G{1}` to obtain the size of the image. This would cause an error because only functions can produce multiple outputs. To obtain `M` and `N` we would use `M = v(1)` and `N = v(2)`. ■

The economy of notation evident in the preceding example becomes even more obvious when the number of outputs is large. One drawback is the loss of clarity in the use of numerical addressing, as opposed to assigning names to the outputs. Using structures helps in this regard.

Structures

Structures are similar to cell arrays in that they allow grouping of a collection of dissimilar data into a single variable. However, unlike cell arrays, in which cells are addressed by numbers, the elements of structures are addressed by user-defined names called *fields*.

EXAMPLE 2.15: Using structures.

■ Continuing with the theme of Example 2.14 will clarify these concepts. Using structures, we write

```
function s = image_stats(f)
s.dm = size(f);
s.AI = mean2(f);
s.AIrows = mean(f, 2);
s.AIcols = mean(f, 1);
```

where `s` is a structure. The fields of the structure in this case are `dm` (a 1×2 vector), `AI` (a scalar), `AIrows` (an $M \times 1$ vector), and `AIcols` (a $1 \times N$ vector), where M and N are the number of rows and columns of the image. Note the use of a dot to separate the structure from its various fields. The field names are arbitrary, but they must begin with a nonnumeric character.

Using the same image as in Example 2.14 and typing `s` and `size(s)` at the prompt gives the following output:

```
>> s =
s =
        dim:  [512 512]
         AI:  1
     AIrows:  [512x1 double]
     AIcols:  [1x512 double]

>> size(s)
ans =
     1     1
```

Note that `s` itself is a scalar, with four fields associated with it in this case.

We see in this example that the logic of the code is the same as before, but the organization of the output data is much clearer. As in the case of cell arrays, the advantage of using structures would become even more evident if we were dealing with a larger number of outputs. ■

The preceding illustration used a single structure. If, instead of one image, we had Q images organized in the form of an $M \times N \times Q$ array, the function would become

```
function s = image_stats(f)
K = size(f);
for k = 1:K(3)
    s(k).dim = size(f(:, :, k));
    s(k).AI = mean2(f(:, :, k));
    s(k).AIrows = mean(f(:, :, k), 2);
    s(k).AIcols = mean(f(:, :, k), 1);
end
```

In other words, structures themselves can be indexed. Although, as with cell arrays, structures can have any number of dimensions, their most common form is a vector, as in the preceding function.

Extracting data from a field requires that the dimensions of both `s` and the field be kept in mind. For example, the following statement extracts all the values of `AIrows` and stores them in `v`:

```
for k = 1:length(s)
    v(:, k) = s(k).AIrows;
end
```

Note that the colon is in the first dimension of `v` and that `k` is in the second because `s` is of dimension $1 \times Q$ and `AIrows` is of dimension $M \times 1$. Thus, because `k` goes

from 1 to Q, `v` is of dimension $M \times Q$. Had we been interested in extracting the values of `AIcols` instead, we would have used `v(k,:)` in the loop.

Square brackets can be used to extract the information into a vector or matrix if the field of a structure contains scalars. For example, suppose that `D.Area` contains the area of each of 20 regions in an image. Writing

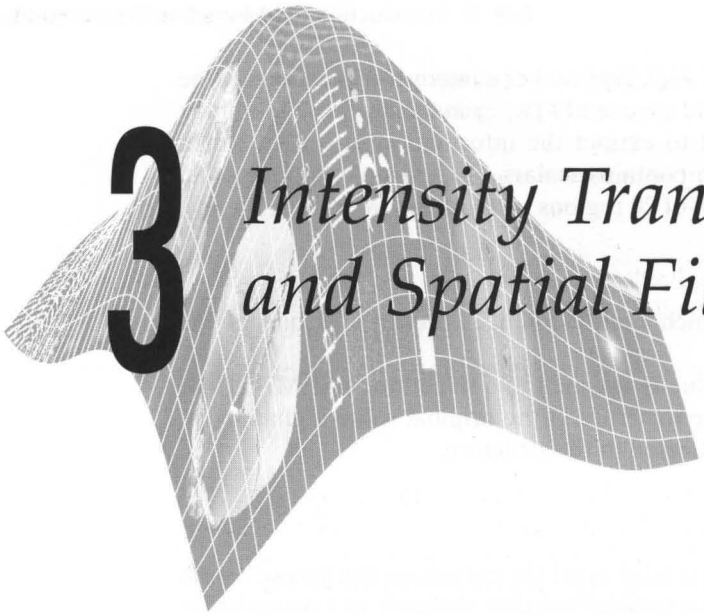
```
>> w = [D.Area];
```

creates a 1×20 vector `w` in which each element is the area of one of the regions.

As with cell arrays, when a value is assigned to a structure field, MATLAB makes a copy of that value in the structure. If the original value is changed at a later time, the change is not reflected in the structure.

Summary

The material in this chapter is the foundation for the discussions that follow. At this point, you should be able to retrieve an image from disk, process it via simple manipulations, display the result, and save it to disk. It is important to note that the key lesson from this chapter is how to combine MATLAB and Image Processing Toolbox functions with programming constructs to generate solutions that expand the capabilities of those functions. In fact, this is the model of how material is presented in the following chapters. By combining standard functions with new code, we show prototypic solutions to a broad spectrum of problems of interest in digital image processing.



3 Intensity Transformations and Spatial Filtering

Preview

The term *spatial domain* refers to the image plane itself, and methods in this category are based on direct manipulation of pixels in an image. In this chapter we focus attention on two important categories of spatial domain processing: *intensity* (*gray-level*) *transformations* and *spatial filtering*. The latter approach sometimes is referred to as *neighborhood processing*, or *spatial convolution*. In the following sections we develop and illustrate MATLAB formulations representative of processing techniques in these two categories. We also introduce the concept of fuzzy image processing and develop several new M-functions for their implementation. In order to carry a consistent theme, most of the examples in this chapter are related to image enhancement. This is a good way to introduce spatial processing because enhancement is highly intuitive and appealing, especially to beginners in the field. As you will see throughout the book, however, these techniques are general in scope and have uses in numerous other branches of digital image processing.

3.1 Background

As noted in the preceding paragraph, spatial domain techniques operate directly on the pixels of an image. The spatial domain processes discussed in this chapter are denoted by the expression

$$g(x, y) = T[f(x, y)]$$

where $f(x, y)$ is the input image, $g(x, y)$ is the output (processed) image, and T is an operator on f defined over a specified neighborhood about point (x, y). In addition, T can operate on a set of images, such as performing the addition of K images for noise reduction.

The principal approach for defining spatial neighborhoods about a point (x, y) is to use a square or rectangular region *centered* at (x, y), as in Fig. 3.1. The center of the region is moved from pixel to pixel starting, say, at the top, left corner, and, as it moves, it encompasses different neighborhoods. Operator T is applied at each location (x, y) to yield the output, g, at that location. Only the pixels in the neighborhood centered at (x, y) are used in computing the value of g at (x, y).

Most of the remainder of this chapter deals with various implementations of the preceding equation. Although this equation is simple conceptually, its computational implementation in MATLAB requires that careful attention be paid to data classes and value ranges.

3.2 Intensity Transformation Functions

The simplest form of the transformation T is when the neighborhood in Fig. 3.1 is of size 1×1 (a single pixel). In this case, the value of g at (x, y) depends only on the intensity of f at that point, and T becomes an *intensity* or *gray-level transformation function*. These two terms are used interchangeably when dealing with monochrome (i.e., gray-scale) images. When dealing with color images, the term *intensity* is used to denote a color image component in certain color spaces, as described in Chapter 7.

Because the output value depends only on the intensity value at a point, and not on a neighborhood of points, intensity transformation functions frequently are written in simplified form as

$$s = T(r)$$

where r denotes the intensity of f and s the intensity of g, both at the same coordinates (x, y) in the images.

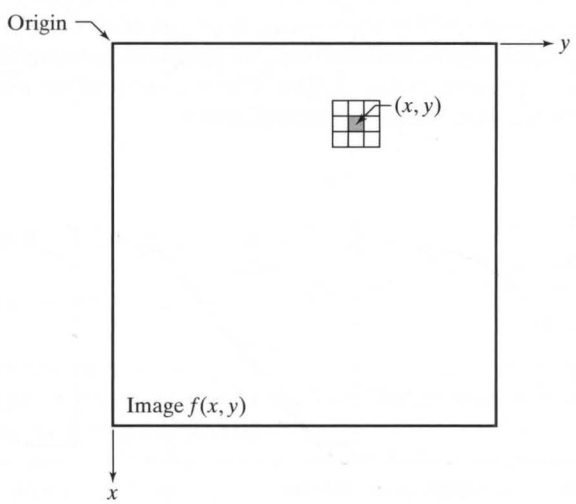
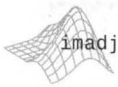


FIGURE 3.1
A neighborhood of size 3×3 centered at point (x, y) in an image.

3.2.1 Functions `imadjust` and `stretchlim`

Function `imadjust` is the basic Image Processing Toolbox function for intensity transformations of gray-scale images. It has the general syntax




```
g = imadjust(f, [low_in high_in], [low_out high_out], gamma)
```

As Fig. 3.2 illustrates, this function maps the intensity values in image `f` to new values in `g`, such that values between `low_in` and `high_in` map to values between `low_out` and `high_out`. Values below `low_in` and above `high_in` are clipped; that is, values below `low_in` map to `low_out`, and those above `high_in` map to `high_out`. The input image can be of class `uint8`, `uint16`, `int16`, `single`, or `double`, and the output image has the same class as the input. All inputs to function `imadjust`, other than `f` and `gamma`, are specified as values between 0 and 1, independently of the class of `f`. If, for example, `f` is of class `uint8`, `imadjust` multiplies the values supplied by 255 to determine the actual values to use. Using the empty matrix (`[ ]`) for `[low_in high_in]` or for `[low_out high_out]` results in the default values `[0 1]`. If `high_out` is less than `low_out`, the output intensity is reversed.

Recall from the discussion in Section 2.7 that function `mat2gray` can be used for converting an image to class `double` and scaling its intensities to the range [0, 1], independently of the class of the input image.

Parameter `gamma` specifies the shape of the curve that maps the intensity values in `f` to create `g`. If `gamma` is less than 1, the mapping is weighted toward higher (brighter) output values, as in Fig. 3.2(a). If `gamma` is greater than 1, the mapping is weighted toward lower (darker) output values. If it is omitted from the function argument, `gamma` defaults to 1 (linear mapping).

EXAMPLE 3.1: Using function `imadjust`.

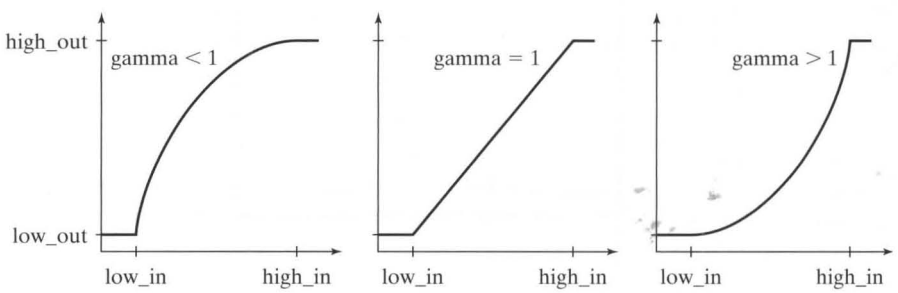
■ Figure 3.3(a) is a digital mammogram image, `f`, showing a small lesion, and Fig. 3.3(b) is the negative image, obtained using the command

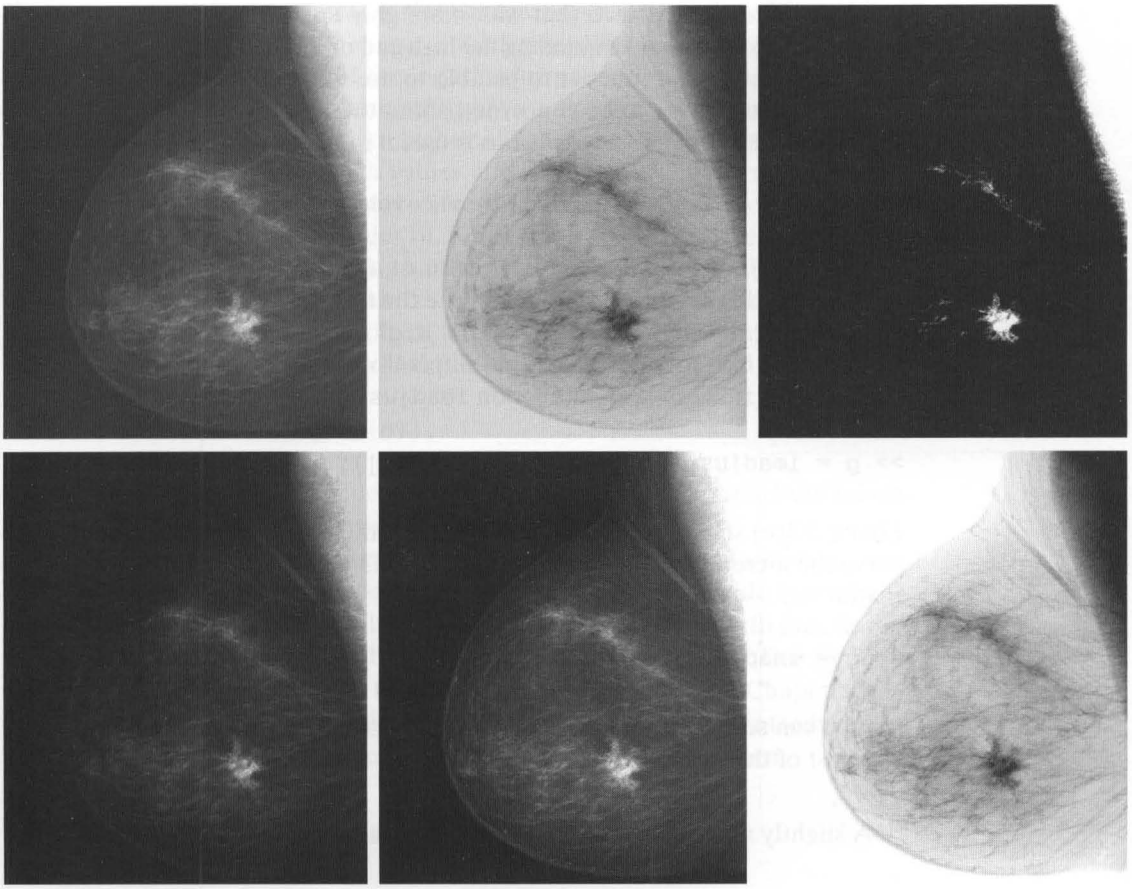
```
>> g1 = imadjust(f, [0 1], [1 0]);
```

This process, which is the digital equivalent of obtaining a photographic negative, is particularly useful for enhancing white or gray detail embedded in a large, predominantly dark region. Note, for example, how much easier it is to analyze the breast tissue in Fig. 3.3(b). The negative of an image can be obtained also with toolbox function `imcomplement`:

a b c

FIGURE 3.2 The various mappings available in function `imadjust`.





a b c
d e f

FIGURE 3.3 (a) Original digital mammogram. (b) Negative image. (c) Result of expanding the intensities in the range [0.5, 0.75]. (d) Result of enhancing the image with `gamma` = 2. (e) and (f) Results of using function `stretchlim` as an automatic input into function `imadjust`. (Original image courtesy of G. E. Medical Systems.)

```
g = imcomplement(f)
```

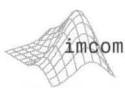


Figure 3.3(c) is the result of using the command

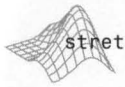
```
>> g2 = imadjust(f, [0.5 0.75], [0 1]);
```

which expands the gray scale interval between 0.5 and 0.75 to the full [0, 1] range. This type of processing is useful for highlighting an intensity band of interest. Finally, using the command

```
>> g3 = imadjust(f, [ ], [ ], 2);
```

produced a result similar to (but with more gray tones than) Fig. 3.3(c) by compressing the low end and expanding the high end of the gray scale [Fig. 3.3(d)].

Sometimes, it is of interest to be able to use function `imadjust` "automatically," without having to be concerned about the low and high parameters discussed above. Function `stretchlim` is useful in that regard; its basic syntax is




```
Low_High = stretchlim(f)
```

where `Low_High` is a two-element vector of a lower and upper limit that can be used to achieve *contrast stretching* (see the following section for a definition of this term). By default, values in `Low_High` specify the intensity levels that saturate the bottom and top 1% of all pixel values in `f`. The result is used in vector `[low_in high_in]` in function `imadjust`, as follows:

```
>> g = imadjust(f, stretchlim(f), [ ]);
```

Figure 3.3(e) shows the result of performing this operation on Fig. 3.3(a). Observe the increase in contrast. Similarly, Fig. 3.3(f) was obtained using the command

```
>> g = imadjust(f, stretchlim(f), [1 0]);
```

As you can see by comparing Figs. 3.3(b) and (f), this operation enhanced the contrast of the negative image. ■

A slightly more general syntax for `stretchlim` is

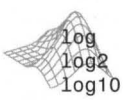
```
Low_High = stretchlim(f, tol)
```

where `tol` is a two-element vector `[low_frac high_frac]` that specifies the fraction of the image to saturate at low and high pixel values.

If `tol` is a scalar, `low_frac = tol`, and `high_frac = 1 - low_frac`; this saturates equal fractions at low and high pixel values. If you omit it from the argument, `tol` defaults to [0.01 0.99], giving a saturation level of 2%. If you choose `tol = 0`, then `Low_High = [min(f(:)) max(f(:))]`.

3.2.2 Logarithmic and Contrast-Stretching Transformations

Logarithmic and contrast-stretching transformations are basic tools for dynamic range manipulation. Logarithm transformations are implemented using the expression



`log`, `log2`, and `log10` are the base e, base 2, and base 10 logarithms, respectively.

```
g = c*log(1 + f)
```

where `c` is a constant and `f` is floating point. The shape of this transformation is similar to the gamma curve in Fig. 3.2(a) with the low values set at 0 and the

high values set to 1 on both scales. Note, however, that the shape of the gamma curve is variable, whereas the shape of the log function is fixed.

One of the principal uses of the log transformation is to compress dynamic range. For example, it is not unusual to have a Fourier spectrum (Chapter 4) with values in the range $[0, 10^6]$ or higher. When displayed on a monitor that is scaled linearly to 8 bits, the high values dominate the display, resulting in lost visual detail in the lower intensity values in the spectrum. By computing the log, a dynamic range on the order of, for example, 10^6, is reduced to approximately 14 [i.e., $\log_e(10^6) = 13.8$], which is much more manageable.

When performing a logarithmic transformation, it is often desirable to bring the resulting compressed values back to the full range of the display. For 8 bits, the easiest way to do this in MATLAB is with the statement

```
>> gs = im2uint8(mat2gray(g));
```

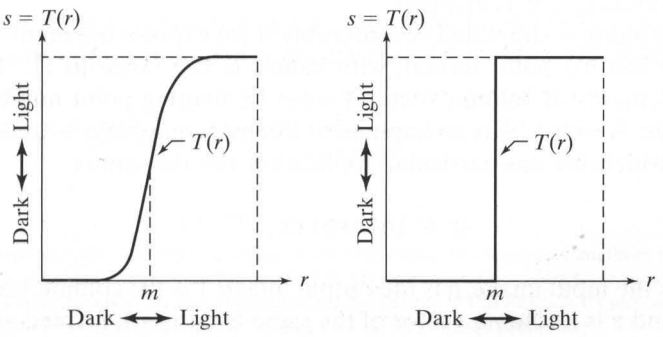
Using `mat2gray` brings the values to the range $[0, 1]$ and using `im2uint8` brings them to the range $[0, 255]$, converting the image to class `uint8`.

The function in Fig. 3.4(a) is called a *contrast-stretching* transformation function because it expands a narrow range of input levels into a wide (stretched) range of output levels. The result is an image of higher contrast. In fact, in the limiting case shown in Fig. 3.4(b), the output is a binary image. This limiting function is called a *thresholding* function, which, as we discuss in Chapter 11, is a simple tool used for image segmentation. Using the notation introduced at the beginning of this section, the function in Fig. 3.4(a) has the form

$$s = T(r) = \frac{1}{1 + (m/r)^E}$$

where r denotes the intensities of the input image, s the corresponding intensity values in the output image, and E controls the slope of the function. This equation is implemented in MATLAB for a floating point image as

```
g = 1./(1 + (m./f).^E)
```

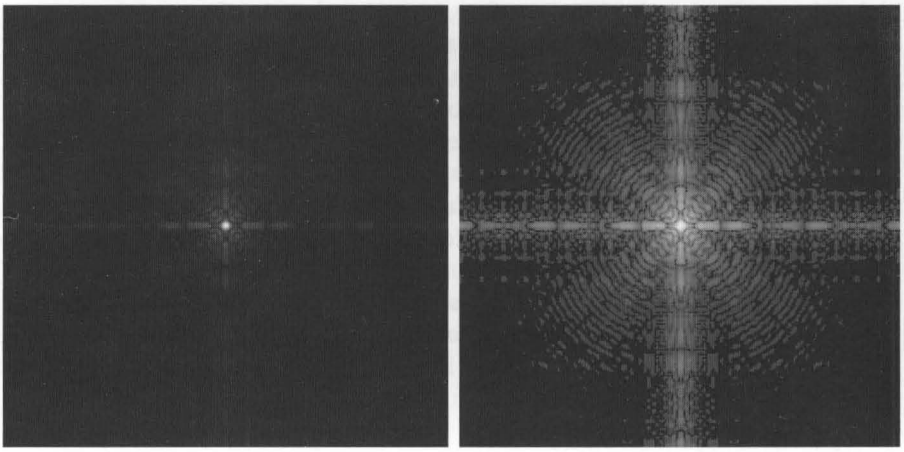


a b

FIGURE 3.4
(a) Contrast-stretching transformation. (b) Thresholding transformation.

a b

FIGURE 3.5
(a) A Fourier spectrum. (b) Result of using a log transformation.



Because the limiting value of `g` is 1, output values cannot exceed the range [0, 1] when working with this type of transformation. The shape in Fig. 3.4(a) was obtained with `E = 20`.

EXAMPLE 3.2: Using a `log` transformation to reduce dynamic range.

■ Figure 3.5(a) is a Fourier spectrum with values in the range 0 to 10^6, displayed on a linearly scaled, 8-bit display system. Figure 3.5(b) shows the result obtained using the commands

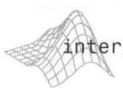
```
>> g = im2uint8(mat2gray(log(1 + double(f))));
>> imshow(g)
```

The visual improvement of `g` over the original image is evident. ■

3.2.3 Specifying Arbitrary Intensity Transformations

Suppose that it is necessary to transform the intensities of an image using a specified transformation function. Let `T` denote a column vector containing the values of the transformation function. For example, in the case of an 8-bit image, `T(1)` is the value to which intensity 0 in the input image is mapped, `T(2)` is the value to which 1 is mapped, and so on, with `T(256)` being the value to which intensity 255 is mapped.

Programming is simplified considerably if we express the input and output images in floating point format, with values in the range [0 1]. This means that all elements of column vector `T` must be floating-point numbers in that same range. A simple way to implement intensity mappings is to use function `interp1` which, for this particular application, has the syntax




```
g = interp1(z, T, f)
```

where `f` is the input image, `g` is the output image, `T` is the column vector just explained, and `z` is a column vector of the same length as `T`, formed as follows:

```
z = linspace(0, 1, numel(T))';
```

See Section 2.8.1 regarding function `linspace`.

For a pixel value in `f`, `interp1` first finds that value in the abscissa (`z`). It then finds (interpolates)[†] the corresponding value in `T` and outputs the interpolated value to `g` in the corresponding pixel location. For example, suppose that `T` is the negative transformation, `T = [1 0]'`. Then, because `T` only has two elements, `z = [0 1]'`. Suppose that a pixel in `f` has the value 0.75. The corresponding pixel in `g` would be assigned the value 0.25. This process is nothing more than the mapping from input to output intensities illustrated in Fig. 3.4(a), but using an arbitrary transformation function $T(r)$. Interpolation is required because we only have a given number of discrete points for T, while r can have any value in the range [0 1].

3.2.4 Some Utility M-Functions for Intensity Transformations

In this section we develop two custom M-functions that incorporate various aspects of the intensity transformations introduced in the previous three sections. We show the details of the code for one of them to illustrate error checking, to introduce ways in which MATLAB functions can be formulated so that they can handle a variable number of inputs and/or outputs, and to show typical code formats used throughout the book. From this point on, detailed code of new M-functions is included in our discussions only when the purpose is to explain specific programming constructs, to illustrate the use of a new MATLAB or Image Processing Toolbox function, or to review concepts introduced earlier. Otherwise, only the syntax of the function is explained, and its code is included in Appendix C. Also, in order to focus on the basic structure of the functions developed in the remainder of the book, this is the last section in which we show extensive use of error checking. The procedures that follow are typical of how error handling is programmed in MATLAB.

Handling a Variable Number of Inputs and/or Outputs

To check the number of arguments input into an M-function we use function `nargin`,

```
n = nargin
```



which returns the actual number of arguments input into the M-function. Similarly, function `nargout` is used in connection with the outputs of an M-function. The syntax is

```
n = nargout
```



[†]Because `interp1` provides interpolated values at discrete points, this function sometimes is interpreted as performing *lookup table* operations. In fact, MATLAB documentation refers to `interp1` parenthetically as a table lookup function. We use a multidimensional version of this function for just that purpose in `approxfcn`, a custom function developed in Section 3.6.4 for fuzzy image processing.

For example, suppose that we execute the following hypothetical M-function at the prompt:

```
>> T = testhv(4, 5);
```

Use of `nargin` within the body of this function would return a 2, while use of `nargout` would return a 1.

Function `nargchk` can be used in the body of an M-function to check if the correct number of arguments was passed. The syntax is




```
msg = nargchk(low, high, number)
```

This function returns the message `Not enough input arguments` if `number` is less than `low` or `Too many input arguments` if `number` is greater than `high`. If `number` is between `low` and `high` (inclusive), `nargchk` returns an empty matrix. A frequent use of function `nargchk` is to stop execution via the `error` function if the incorrect number of arguments is input. The number of actual input arguments is determined by the `nargin` function. For example, consider the following code fragment:

```
function G = testhv2(x, y, z)
  :
error(nargchk(2, 3, nargin));
  :
```

Typing

```
>> testhv2(6);
```

which only has one input argument would produce the error

```
Not enough input arguments.
```

and execution would terminate.

It is useful to be able to write functions in which the number of input and/or output arguments is variable. For this, we use the variables `varargin` and `varargout`. In the declaration, `varargin` and `varargout` must be lowercase. For example,




```
function [m, n] = testhv3(varargin)
```

accepts a variable number of inputs into function `testhv3.m`, and

```
function [varargout] = testhv4(m, n, p)
```

returns a variable number of outputs from function `testhv4`. If function `testhv3` had, say, one fixed input argument, `x`, followed by a variable number of input arguments, then

```
function [m, n] = testhv3(x, varargin)
```

would cause `varargin` to start with the second input argument supplied by the user when the function is called. Similar comments apply to `varargout`. It is acceptable to have a function in which both the number of input and output arguments is variable.

When `varargin` is used as the input argument of a function, MATLAB sets it to a cell array (see Section 2.10.7) that contains the arguments provided by the user. Because `varargin` is a cell array, an important aspect of this arrangement is that the call to the function can contain a mixed set of inputs. For example, assuming that the code of our hypothetical function `testhv3` is equipped to handle it, a perfectly acceptable syntax having a mixed set of inputs could be

```
>> [m, n] = testhv3(f, [0  0.5  1.5], A, 'label');
```

where `f` is an image, the next argument is a row vector of length 3, `A` is a matrix, and `'label'` is a character string. This is a powerful feature that can be used to simplify the structure of functions requiring a variety of different inputs. Similar comments apply to `varargout`.

Another M-Function for Intensity Transformations

In this section we develop a function that computes the following transformation functions: negative, log, gamma and contrast stretching. These transformations were selected because we will need them later, and also to illustrate the mechanics involved in writing an M-function for intensity transformations. In writing this function we use function `tofloat`,

```
[g, revertclass] = tofloat(f)
```

introduced in Section 2.7. Recall from that discussion that this function converts an image of class `logical`, `uint8`, `uint16`, or `int16` to class `single`, applying the appropriate scale factor. If `f` is of class `double` or `single`, then `g = f`; also, recall that `revertclass` is a function handle that can be used to covert the output back to the same class as `f`.

Note in the following M-function, which we call `intrans`, how function options are formatted in the Help section of the code, how a variable number of inputs is handled, how error checking is interleaved in the code, and how the class of the output image is matched to the class of the input. Keep in mind when studying the following code that `varargin` is a cell array, so its elements are selected by using curly braces.

intrans

```
function g = intrans(f, method, varargin)
%INTRANS Performs intensity (gray-level) transformations.
%   G = INTRANS(F, 'neg') computes the negative of input image F.
%
%   G = INTRANS(F, 'log', C, CLASS) computes C*log(1 + F) and
```

```
%   multiplies the result by (positive) constant C. If the last two
%   parameters are omitted, C defaults to 1. Because the log is used
%   frequently to display Fourier spectra, parameter CLASS offers
%   the option to specify the class of the output as 'uint8' or
%   'uint16'. If parameter CLASS is omitted, the output is of the
%   same class as the input.
%
%   G = INTRANS(F, 'gamma', GAM) performs a gamma transformation on
%   the input image using parameter GAM (a required input).
%
%   G = INTRANS(F, 'stretch', M, E) computes a contrast-stretching
%   transformation using the expression 1./(1 + (M./F).^E).
%   Parameter M must be in the range [0, 1]. The default value for
%   M is mean2(tofloat(F)), and the default value for E is 4.
%
%   G = INTRANS(F, 'specified', TXFUN) performs the intensity
%   transformation s = TXFUN(r) where r are input intensities, s are
%   output intensities, and TXFUN is an intensity transformation
%   (mapping) function, expressed as a vector with values in the
%   range [0, 1]. TXFUN must have at least two values.
%
%   For the 'neg', 'gamma', 'stretch' and 'specified'
%   transformations, floating-point input images whose values are
%   outside the range [0, 1] are scaled first using MAT2GRAY. Other
%   images are converted to floating point using TOFLOAT. For the
%   'log' transformation,floating-point images are transformed
%   without being scaled; other images are converted to floating
%   point first using TOFLOAT.
%
%   The output is of the same class as the input, except if a
%   different class is specified for the 'log' option.

% Verify the correct number of inputs.
error(nargchk(2, 4, nargin))

if strcmp(method, 'log')
   % The log transform handles image classes differently than the
   % other transforms, so let the logTransform function handle that
   % and then return.
   g = logTransform(f, varargin{:});
   return;
end

% If f is floating point, check to see if it is in the range [0 1].
% If it is not, force it to be using function mat2gray.
if isfloat(f) && (max(f(:)) > 1 || min(f(:)) < 0)
   f = mat2gray(f);
end
[f, revertclass] = tofloat(f); %Store class of f for use later.

% Perform the intensity transformation specified.
```

```
switch method
case 'neg'
   g = imcomplement(f);

case 'gamma'
   g = gammaTransform(f, varargin{:});

case 'stretch'
   g = stretchTransform(f, varargin{:});

case 'specified'
   g = spcfiedTransform(f, varargin{:});

otherwise
   error('Unknown enhancement method.')
end

% Convert to the class of the input image.
g = revertclass(g);

%-----------------------------------------------------------------%
function g = gammaTransform(f, gamma)
g = imadjust(f, [ ], [ ], gamma);

%-----------------------------------------------------------------%
function g = stretchTransform(f, varargin)
if isempty(varargin)
   % Use defaults.
   m = mean2(f);
   E = 4.0;
elseif length(varargin) == 2
   m = varargin{1};
   E = varargin{2};
else
   error('Incorrect number of inputs for the stretch method.')
end
g = 1./(1 + (m./f).^E);

%-----------------------------------------------------------------%
function g = spcfiedTransform(f, txfun)
% f is floating point with values in the range [0 1].
txfun = txfun(:); % Force it to be a column vector.
if any(txfun) > 1 || any(txfun) <= 0
   error('All elements of txfun must be in the range [0 1].')
end
T = txfun;
X = linspace(0, 1, numel(T))';
g = interp1(X, T, f);

%-----------------------------------------------------------------%
function g = logTransform(f, varargin)
```

```
[f, revertclass] = tofloat(f);
if numel(varargin) >= 2
   if strcmp(varargin{2}, 'uint8')
      revertclass = @im2uint8;
   elseif strcmp(varargin{2}, 'uint16')
      revertclass = @im2uint16;
   else
      error('Unsupported CLASS option for ''log'' method.')
   end
end
if numel(varargin) < 1
   % Set default for C.
   C = 1;
else
   C = varargin{1};
end
g = C * (log(1 + f));
g = revertclass(g);
```

EXAMPLE 3.3: Illustration of function `intrans`.

■ As an illustration of function `intrans`, consider the image in Fig. 3.6(a), which is an ideal candidate for contrast stretching to enhance the skeletal structure. The result in Fig. 3.6(b) was obtained with the following call to `intrans`:

```
>> g = intrans(f, 'stretch', mean2(tofloat(f)), 0.9);
>> figure, imshow(g)
```

Note how function `mean2` was used to compute the mean value of `f` directly inside the function call. The resulting value was used for `m`. Image `f` was converted to floating point using `tofloat` in order to scale its values to the range [0, 1] so that the mean would also be in this range, as required for input `m`. The value of `E` was determined interactively. ■

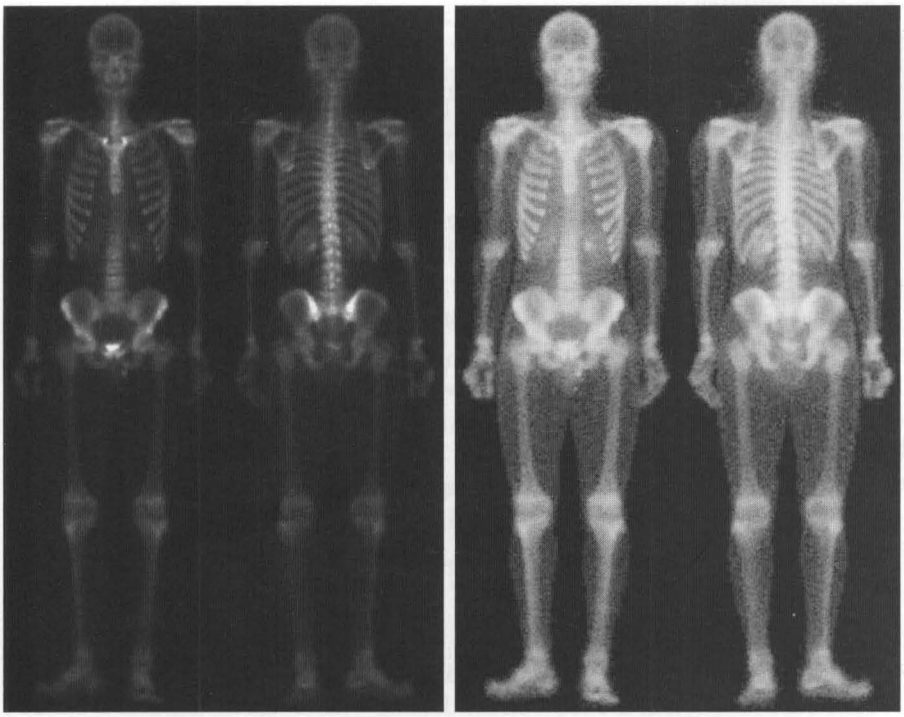
An M-Function for Intensity Scaling

When working with images, computations that result in pixel values that span a wide negative to positive range are common. While this presents no problems during intermediate computations, it does become an issue when we want to use an 8-bit or 16-bit format for saving or viewing an image, in which case it usually is desirable to scale the image to the full, maximum range, [0, 255] or [0, 65535]. The following custom M-function, which we call `gscale`, accomplishes this. In addition, the function can map the output levels to a specified range. The code for this function does not include any new concepts so we do not include it here. See Appendix C for the listing.

The syntax of function `gscale` is

`gscale`

```
g = gscale(f, method, low, high)
```



a b

FIGURE 3.6
(a) Bone scan image. (b) Image enhanced using a contrast-stretching transformation. (Original image courtesy of G. E. Medical Systems.)

where `f` is the image to be scaled. Valid values for `method` are `'full8'` (the default), which scales the output to the full range [0, 255], and `'full16'`, which scales the output to the full range [0, 65535]. If included, parameters `low` and `high` are ignored in these two conversions. A third valid value of `method` is `'minmax'`, in which case parameters `low` and `high`, both in the range [0, 1], must be provided. If `'minmax'` is selected, the levels are mapped to the range `[low, high]`. Although these values are specified in the range [0, 1], the program performs the proper scaling, depending on the class of the input, and then converts the output to the same class as the input. For example, if `f` is of class `uint8` and we specify `'minmax'` with the range [0, 0.5], the output also will be of class `uint8`, with values in the range [0, 128]. If `f` is floating point and its range of values is outside the range [0, 1], the program converts it to this range before proceeding. Function `gscale` is used in numerous places throughout the book.

3.3 Histogram Processing and Function Plotting

Intensity transformation functions based on information extracted from image intensity histograms play a central role in image processing, in areas such as enhancement, compression, segmentation, and description. The focus of this section is on obtaining, plotting, and using histograms for image enhancement. Other applications of histograms are discussed in later chapters.

See Section 4.5.3 for a discussion of 2-D plotting techniques.

3.3.1 Generating and Plotting Image Histograms

The histogram of a digital image with L total possible intensity levels in the range $[0, G]$ is defined as the discrete function

$$h(r_k) = n_k$$

where r_k is the kth intensity level in the interval $[0, G]$ and n_k is the number of pixels in the image whose intensity level is r_k. The value of G is 255 for images of class `uint8`, 65535 for images of class `uint16`, and 1.0 for floating point images. Note that $G = L - 1$ for images of class `uint8` and `uint16`.

Sometimes it is necessary to work with *normalized* histograms, obtained simply by dividing all elements of $h(r_k)$ by the total number of pixels in the image, which we denote by n:

$$p(r_k) = \frac{h(r_k)}{n} = \frac{n_k}{n}$$

where, for integer images, $k = 0, 1, 2, \ldots, L - 1$. From basic probability, we recognize $p(r_k)$ as an estimate of the probability of occurrence of intensity level r_k.

The core function in the toolbox for dealing with image histograms is `imhist`, with the basic syntax:




```
h = imhist(f, b)
```

where `f` is the input image, `h` is its histogram, and `b` is the number of bins used in forming the histogram (if `b` is not included in the argument, `b = 256` is used by default). A bin is simply a subdivision of the intensity scale. For example, if we are working with `uint8` images and we let `b = 2`, then the intensity scale is subdivided into two ranges: 0 to 127 and 128 to 255. The resulting histogram will have two values: `h(1)`, equal to the number of pixels in the image with values in the interval [0, 127] and `h(2)`, equal to the number of pixels with values in the interval [128, 255]. We obtain the normalized histogram by using the expression

```
p = imhist(f, b)/numel(f)
```

Recall from Section 2.10.3 that function `numel(f)` gives the number of elements in array `f` (i.e., the number of pixels in the image).

EXAMPLE 3.4: Computing and plotting image histograms.

■ Consider the image, `f`, from Fig. 3.3(a). The simplest way to plot its histogram on the screen is to use `imhist` with no output specified:

```
>> imhist(f);
```

Figure 3.7(a) shows the result. This is the histogram display default in the toolbox. However, there are many other ways to plot a histogram, and we take this opportunity to explain some of the plotting options in MATLAB that are representative of those used in image processing applications.

Histograms can be plotted also using *bar* graphs. For this purpose we can use the function

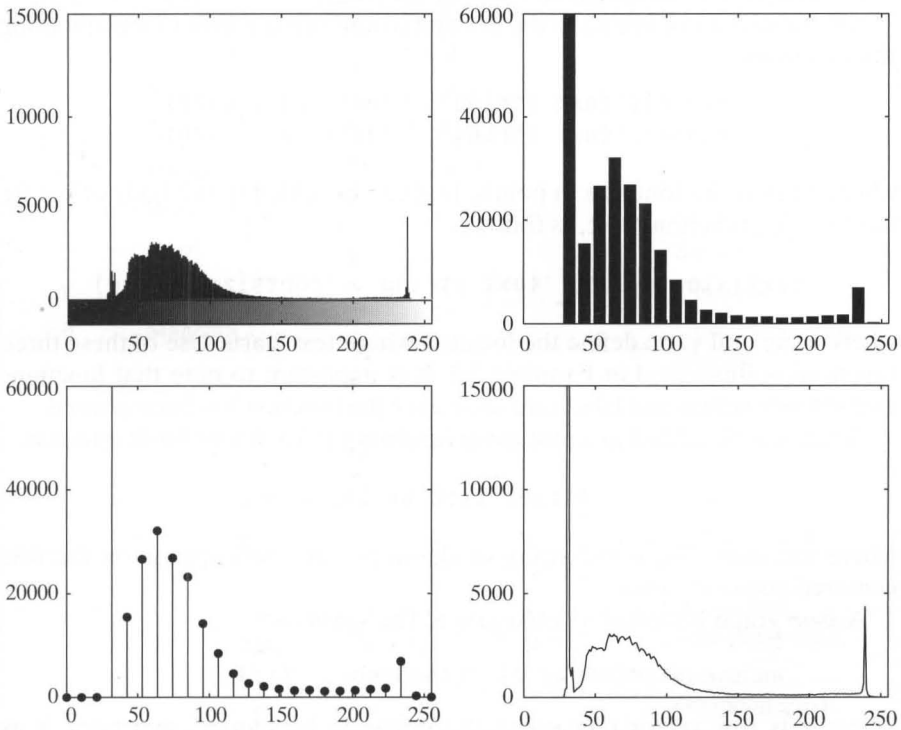
```
bar(horz, z, width)
```



where `z` is a row vector containing the points to be plotted, `horz` is a vector of the same dimension as `z` that contains the increments of the horizontal scale, and `width` is a number between 0 and 1. In other words, the values of `horz` give the horizontal increments and the values of `z` are the corresponding vertical values. If `horz` is omitted, the horizontal axis is divided in units from 0 to `length(z)`. When `width` is 1, the bars touch; when it is 0, the bars are vertical lines. The default value is 0.8. When plotting a bar graph, it is customary to reduce the resolution of the horizontal axis by dividing it into bands.

The following commands produce a bar graph, with the horizontal axis divided into groups of approximately 10 levels:

```
>> h = imhist(f, 25);
>> horz = linspace(0, 255, 25);
```



a b
c d

FIGURE 3.7 Various ways to plot an image histogram. (a) `imhist`, (b) `bar`, (c) `stem`, (d) `plot`.




```
>> bar(horz, h)
>> axis([0 255 0 60000])
>> set(gca, 'xtick', 0:50:255)
>> set(gca, 'ytick', 0:20000:60000)
```

Figure 3.7(b) shows the result. The narrow peak located at the high end of the intensity scale in Fig. 3.7(a) is lower in the bar graph because larger horizontal increments were used in that graph. The vertical scale spans a wider range of values than for the full histogram in Fig. 3.7(a) because the height of each bar is determined by all pixels in a range, rather than by all pixels with a single value.

The fourth statement in the preceding code was used to expand the lower range of the vertical axis for visual analysis, and to set the horizontal axis to the same range as in Fig. 3.7. One of the `axis` function syntax forms is



`axis ij` places the origin of the axis system on the top left. This is the default is when superimposing axes on images. As we show in Example 5.12, sometimes it is useful to have the origin on the bottom left. Using `axis xy` does that.

```
axis([horzmin  horzmax  vertmin  vertmax])
```

which sets the minimum and maximum values in the horizontal and vertical axes. In the last two statements, `gca` means "get current axis" (i.e., the axes of the figure last displayed), and `xtick` and `ytick` set the horizontal and vertical axes ticks in the intervals shown. Another syntax used frequently is

```
axis tight
```

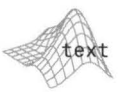
which sets the axis limits to the range of the data.

Axis labels can be added to the horizontal and vertical axes of a graph using the functions




```
xlabel('text string', 'fontsize', size)
ylabel('text string', 'fontsize', size)
```

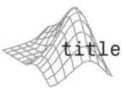
where `size` is the font size in points. Text can be added to the body of the figure by using function `text`, as follows:




```
text(xloc, yloc, 'text string', 'fontsize', size)
```

where `xloc` and `yloc` define the location where text starts. Use of these three functions is illustrated in Example 3.4. It is important to note that functions that set axis values and labels are used *after* the function has been plotted.

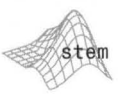
A title can be added to a plot using function `title`, whose basic syntax is




```
title('titlestring')
```

where `titlestring` is the string of characters that will appear on the title, centered above the plot.

A *stem* graph is similar to a bar graph. The syntax is




```
stem(horz, z, 'LineSpec', 'fill')
```

where `z` is row vector containing the points to be plotted, and `horz` is as

described for function `bar`. If `horz` is omitted, the horizontal axis is divided in units from 0 to `length(z)`, as before.

The argument,

```
LineSpec
```

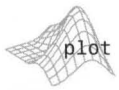
is a triplet of values from Table 3.1. For example, `stem(horz, h, 'r--p')` produces a stem plot where the lines and markers are red, the lines are dashed, and the markers are five-point stars. If `fill` is used, the marker is filled with the color specified in the first element of the triplet. The default color is `blue`, the line default is `solid`, and the default marker is a `circle`. The stem graph in Fig. 3.7(c) was obtained using the statements

```
>> h = imhist(f, 25);
>> horz = linspace(0, 255, 25);
>> stem(horz, h, 'fill')
>> axis([0 255 0 60000])
>> set(gca, 'xtick', [0:50:255])
>> set(gca, 'ytick', [0:20000:60000])
```

Next, we consider function `plot`, which plots a set of points by linking them with straight lines. The syntax is

Color Specifiers		Line Specifiers		Marker Specifiers	
Symbol	**Color**	**Symbol**	**Line Style**	**Symbol**	**Marker**
k	Black	-	Solid	+	Plus sign
w	White	--	Dashed	o	Circle
r	Red	:	Dotted	*	Asterisk
g	Green	-.	Dash-dot	.	Point
b	Blue			x	Cross
c	Cyan			s	Square
y	Yellow			d	Diamond
m	Magenta			^	Upward-pointing triangle
				v	Downward-pointing triangle
				>	Right-pointing triangle
				<	Left-pointing triangle
				p	Pentagram (five-point star)
				h	Hexagram (six-point star)

TABLE 3.1
Color, line, and marker specifiers for use in functions `stem` and `plot`.



See the `plot` help page for additional options available for this function.

```
plot(horz, z, 'LineSpec')
```

where the arguments are as defined previously for stem plots. As in `stem`, the attributes in `plot` are specified as a triplet. The defaults for `plot` are solid blue lines with no markers. If a triplet is specified in which the middle value is blank (or omitted), no lines are plotted. As before, if `horz` is omitted, the horizontal axis is divided in units from 0 to `length(z)`.

The plot in Fig. 3.7(d) was obtained using the following statements:

```
>> hc = imhist(f);
>> plot(hc) % Use the default values.
>> axis([0 255 0 15000])
>> set(gca, 'xtick', [0:50:255])
>> set(gca, 'ytick', [0:2000:15000])
```

Function `plot` is used frequently to display transformation functions (see Example 3.5). ■

Plot defaults are useful for superimposing markers on an image. For example, to place green asterisks at points given in vectors `x` and `y` in an image, `f`, we use:

```
>> imshow(f)
>> hold on
>> plot(y(:),x(:),'g*')
```

where the order of `y(:)` and `x(:)` is reversed to compensate for the fact that image and plot coordinate systems are different in MATLAB. Command `hold on` is explained below.

In the preceding discussion axis limits and tick marks were set manually. To set the limits and ticks automatically, use functions `ylim` and `xlim`, which, for our purposes here, have the syntax forms

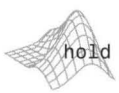



```
ylim('auto')
xlim('auto')
```

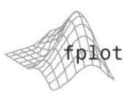
Among other possible variations of the syntax for these two functions (see the help documentation for details), there is a manual option, given by

```
ylim([ymin  ymax])
xlim([xmin  xmax])
```

which allows manual specification of the limits. If the limits are specified for only one axis, the limits on the other axis are set to `'auto'` by default. We use these functions in the following section. Typing `hold on` at the prompt retains the current plot and certain axes properties so that subsequent graphing commands add to the existing graph.



Another plotting function that is particularly useful when dealing with function handles (see Sections 2.10.4 and 2.10.5) is function `fplot`. The basic syntax is



See the help page for `fplot` for a discussion of additional syntax forms.

```
fplot(fhandle, limits, 'LineSpec')
```

where `fhandle` is a function handle, and `limits` is a vector specifying the x-axis limits, `[xmin  xmax]`. You will recall from the discussion of function `timeit` in Section 2.10.5 that using function handles allows the syntax of the underlying function to be independent of the parameters of the function to be processed (plotted in this case). For example, to plot the hyperbolic tangent function, `tanh`, in the range `[-2 2]` using a dotted line we write

```
>> fhandle = @tanh;
>> fplot(fhandle, [-2 2], ':')
```

Function `fplot` uses an automatic, adaptive increment control scheme to produce a representative graph, concentrating more detail where the rate of change is the greatest. Thus, only the plotting limits have to be specified by the user. While this simplifies plotting tasks, the automatic feature can at times yield unexpected results. For example, if a function is initially 0 for an appreciable interval, it is possible for `fplot` to assume that the function is zero and just plot 0 for the entire interval. In cases such as this, you can specify a minimum number of points for the function to plot. The syntax is

```
fplot(fhandle, limits, 'LineSpec', n)
```

Specifying `n >= 1` forces `fplot` to plot the function with a minimum of `n + 1` points, using a step size of `(1/n)*(upper_lim - lower_lim)`, where upper and lower refer to the upper and lower limits specified in `limits`.

3.3.2 Histogram Equalization

Assume for a moment that intensity levels are continuous quantities normalized to the range [0, 1], and let $p_r(r)$ denote the probability density function (PDF) of the intensity levels in a given image, where the subscript is used for differentiating between the PDFs of the input and output images. Suppose that we perform the following transformation on the input levels to obtain output (processed) intensity levels, s,

$$s = T(r) = \int_0^r p_r(w)\,dw$$

where w is a dummy variable of integration. It can be shown (Gonzalez and Woods [2008]) that the probability density function of the output levels is *uniform*; that is,

$$p_s(s) = \begin{cases} 1 & \text{for } 0 \le s \le 1 \\ 0 & \text{otherwise} \end{cases}$$

In other words, the preceding transformation generates an image whose intensity levels are equally likely, and, in addition, cover the entire range [0, 1]. The net result of this intensity-level *equalization* process is an image with increased dynamic range, which will tend to have higher contrast. Note that the transformation function is really nothing more than the cumulative distribution function (CDF).

When dealing with discrete quantities we work with histograms and call the preceding technique *histogram equalization*, although, in general, the histogram of the processed image will not be uniform, due to the discrete nature of the variables. With reference to the discussion in Section 3.3.1, let $p_r(r_j)$ for $j = 0, 1, 2, \ldots, L - 1$, denote the histogram associated with the intensity levels

of a given image, and recall that the values in a normalized histogram are approximations to the probability of occurrence of each intensity level in the image. For discrete quantities we work with summations, and the equalization transformation becomes

$$\begin{aligned} s_k &= T(r_k) \\ &= \sum_{j=0}^{k} p_r(r_j) \\ &= \sum_{j=0}^{k} \frac{n_j}{n} \end{aligned}$$

for $k = 0, 1, 2, \ldots, L-1$, where s_k is the intensity value in the output (processed) image corresponding to value r_k in the input image.

Histogram equalization is implemented in the toolbox by function `histeq`, which has the syntax




```
g = histeq(f, nlev)
```

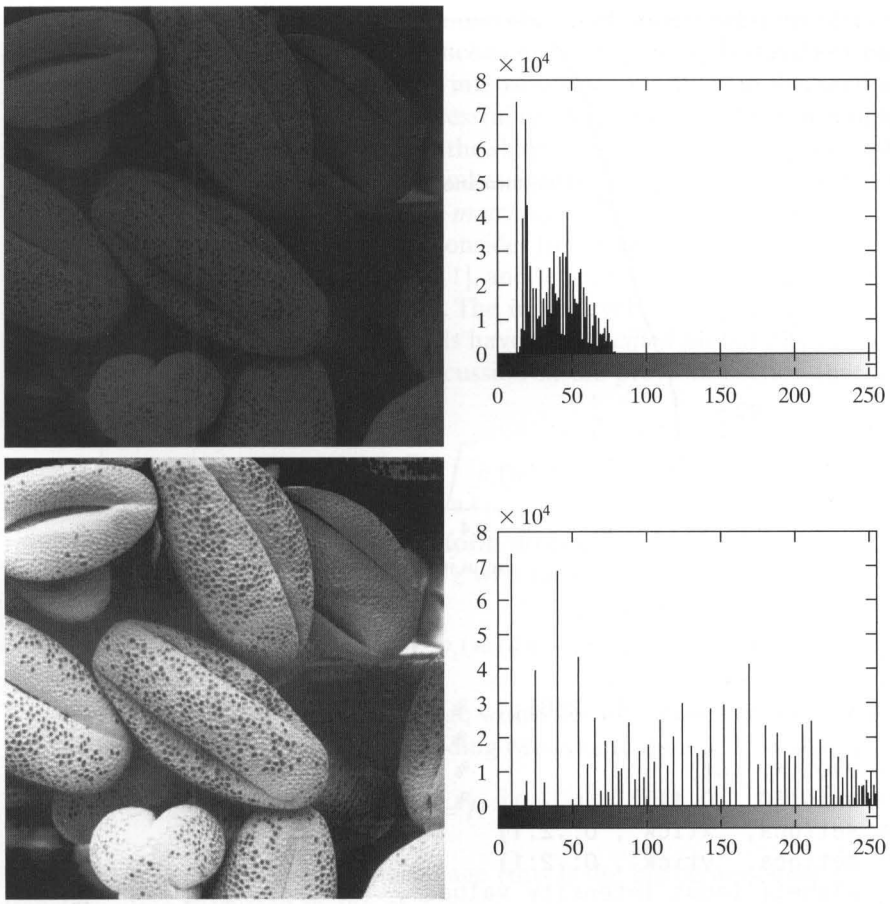
where `f` is the input image and `nlev` is the number of intensity levels specified for the output image. If `nlev` is equal to L (the total number of *possible* levels in the input image), then `histeq` implements the transformation function directly. If `nlev` is less than L, then `histeq` attempts to distribute the levels so that they will approximate a flat histogram. Unlike `imhist`, the default value in `histeq` is `nlev = 64`. For the most part, we use the maximum possible number of levels (generally 256) for `nlev` because this produces a true implementation of the histogram-equalization method just described.

EXAMPLE 3.5: Histogram equalization.

■ Figure 3.8(a) is an electron microscope image of pollen, magnified approximately 700 times. In terms of needed enhancement, the most important features of this image are that it is dark and has a low dynamic range. These characteristics are evident in the histogram in Fig. 3.8(b), in which the dark nature of the image causes the histogram to be biased toward the dark end of the gray scale. The low dynamic range is evident from the fact that the histogram is narrow with respect to the entire gray scale. Letting `f` denote the input image, the following sequence of steps produced Figs. 3.8(a) through (d):

```
>> imshow(f); % Fig. 3.8(a).
>> figure, imhist(f) % Fig. 3.8(b).
>> ylim('auto')
>> g = histeq(f, 256);
>> figure, imshow(g) % Fig. 3.8(c).
>> figure, imhist(g) % Fig. 3.8(d).
>> ylim('auto')
```

The image in Fig. 3.8(c) is the histogram-equalized result. The improvements in average intensity and contrast are evident. These features also are



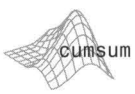
a b
c d

FIGURE 3.8 Illustration of histogram equalization. (a) Input image, and (b) its histogram. (c) Histogram-equalized image, and (d) its histogram. The improvement between (a) and (c) is evident. (Original image courtesy of Dr. Roger Heady, Research School of Biological Sciences, Australian National University, Canberra.)

evident in the histogram of this image, shown in Fig. 3.8(d). The increase in contrast is due to the considerable spread of the histogram over the entire intensity scale. The increase in overall intensity is due to the fact that the average intensity level in the histogram of the equalized image is higher (lighter) than the original. Although the histogram-equalization method just discussed does not produce a flat histogram, it has the desired characteristic of being able to increase the dynamic range of the intensity levels in an image.

As noted earlier, the transformation function used in histogram equalization is the cumulative sum of normalized histogram values. We can use function `cumsum` to obtain the transformation function, as follows:

If `A` is a vector, `B = cumsum(A)` gives the sum of its elements. If `A` is a higher-dimensional array, then `B = cumsum(A, dim)` gives the sum along the dimension specified by `dim`.




```
>> hnorm = imhist(f)./numel(f); % Normalized histogram.
>> cdf = cumsum(hnorm); % CDF.
```

A plot of `cdf`, shown in Fig. 3.9, was obtained using the following commands:

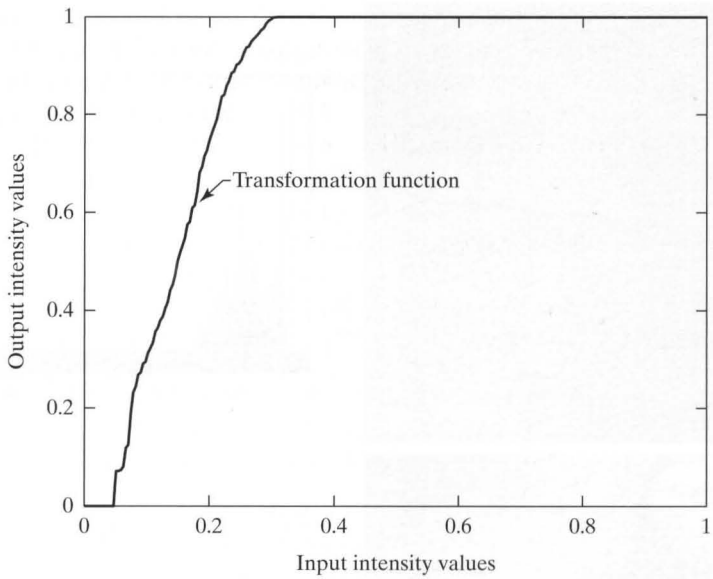
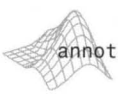


FIGURE 3.9 Transformation function used to map the intensity values from the input image in Fig. 3.7(a) to the values of the output image in Fig. 3.7(c).

```
>> x = linspace(0, 1, 256);    % Intervals for [0,1] horiz
                               % scale.
>> plot(x, cdf)                % Plot cdf vs. x.
>> axis([0 1 0 1]);            % Scale, settings, and labels:
>> set(gca, 'xtick', 0:.2:1)
>> set(gca, 'ytick', 0:.2:1)
>> xlabel('Input intensity values', 'fontsize', 9)
>> ylabel('Output intensity values', 'fontsize', 9)
```

The text in the body of the graph was inserted using the `TextBox` and `Arrow` commands from the **Insert** menu in the MATLAB figure window containing the plot. You can use function `annotation` to write code that inserts items such as text boxes and arrows on graphs, but the **Insert** menu is considerably easier to use.



See the help page for this function for details on how to use it.

You can see by looking at the histograms in Fig. 3.8 that the transformation function in Fig. 3.9 maps a narrow range of intensity levels in the lower end of the input intensity scale to the full intensity range in the output image. The improvement in image contrast is evident by comparing the input and output images in Fig. 3.8. ■

3.3.3 Histogram Matching (Specification)

Histogram equalization produces a transformation function that is adaptive, in the sense that it is based on the histogram of a given image. However, once the transformation function for an image has been computed, it does not change

unless the histogram of the image changes. As noted in the previous section, histogram equalization achieves enhancement by spreading the levels of the input image over a wider range of the intensity scale. We show in this section that this does not always lead to a successful result. In particular, it is useful in some applications to be able to specify the shape of the histogram that we wish the processed image to have. The method used to generate an image that has a specified histogram is called *histogram matching* or *histogram specification*.

The method is simple in principle. Consider for a moment continuous levels that are normalized to the interval [0, 1], and let r and z denote the intensity levels of the input and output images. The input levels have probability density function $p_r(r)$ and the output levels have the *specified* probability density function $p_z(z)$. We know from the discussion in the previous section that he transformation

$$s = T(r) = \int_0^r p_r(w)\,dw$$

results in intensity levels, s, with a uniform probability density function $p_s(s)$. Suppose now that we define a variable z with the property

$$H(z) = \int_0^z p_z(w)\,dw = s$$

Keep in mind that we are after an image with intensity levels, z, that have the specified density $p_z(z)$. From the preceding two equations, it follows that

$$z = H^{-1}(s) = H^{-1}\left[T(r)\right]$$

We can find $T(r)$ from the input image (this is the histogram-equalization transformation discussed in the previous section), so it follows that we can use the preceding equation to find the transformed levels z whose density is the specified $p_z(z)$ provided that we can find H^{-1}. When working with discrete variables, we can guarantee that the inverse of H exists if $p(z_k)$ is a valid histogram (i.e., it has unit area and all its values are nonnegative), *and* none of its components is zero [i.e., no bin of $p(z_k)$ is empty]. As in histogram equalization, the discrete implementation of the preceding method only yields an approximation to the specified histogram.

The toolbox implements histogram matching using the following syntax in `histeq`:

```
g = histeq(f, hspec)
```

where `f` is the input image, `hspec` is the specified histogram (a row vector of specified values), and `g` is the output image, whose histogram approximates the specified histogram, `hspec`. This vector should contain integer counts corresponding to equally spaced bins. A property of `histeq` is that the histogram of `g` generally better matches `hspec` when `length(hspec)` is much smaller than the number of intensity levels in `f`.

EXAMPLE 3.6: Histogram matching.

■ Figure 3.10(a) shows an image, `f`, of the Mars moon, Phobos, and Fig. 3.10(b) shows its histogram, obtained using `imhist(f)`. The image is dominated by large, dark areas, resulting in a histogram characterized by a large concentration of pixels in the dark end of the gray scale. At first glance, one might conclude that histogram equalization would be a good approach to enhance this image, so that details in the dark areas become more visible. However, the result in Fig. 3.10(c), obtained using the command

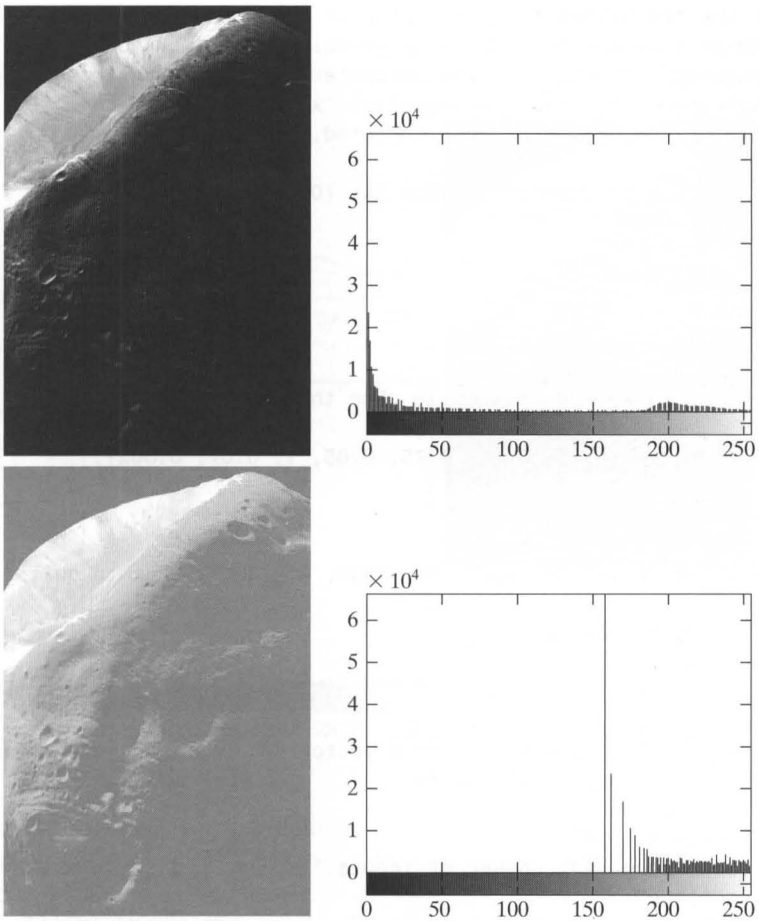
```
>> f1 = histeq(f, 256);
```

shows that histogram equalization in fact produced an image with a "washed-out" appearance—not a particularly good result in this case. The reason for this can be seen by studying the histogram of the equalized image, shown in Fig. 3.10(d). Here, we see that the intensity levels have been shifted to the upper one-half of the gray scale, thus giving the image the low-contrast, washed-out appearance mentioned above. The cause of the shift is the large concentration of dark components at or near 0 in the original histogram. The cumulative transformation function obtained from this histogram is steep, thus mapping the large concentration of pixels in the low end of the gray scale to the high end of the scale.

One possibility for remedying this situation is to use histogram matching, with the desired histogram having a lesser concentration of components in the low end of the gray scale, and maintaining the general shape of the histogram of the original image. We note from Fig. 3.10(b) that the histogram is basically bimodal, with one large mode at the origin, and another, smaller, mode at the high end of the gray scale. These types of histograms can be modeled, for example, by using multimodal Gaussian functions. The following M-function computes a bimodal Gaussian function normalized to unit area, so it can be used as a specified histogram.

twomodegauss

```
function p = twomodegauss(m1, sig1, m2, sig2, A1, A2, k)
%TWOMODEGAUSS Generates a two-mode Gaussian function.
%   P = TWOMODEGAUSS(M1, SIG1, M2, SIG2, A1, A2, K) generates a
%   two-mode, Gaussian-like function in the interval [0, 1]. P is a
%   256-element vector normalized so that SUM(P) = 1. The mean and
%   standard deviation of the modes are (M1, SIG1) and (M2, SIG2),
%   respectively. A1 and A2 are the amplitude values of the two
%   modes. Since the output is normalized, only the relative values
%   of A1 and A2 are important. K is an offset value that raises the
%   "floor" of the function. A good set of values to try is M1 =
%   0.15, SIG1 = 0.05, M2 = 0.75, SIG2 = 0.05, A1 = 1, A2 = 0.07,
%   and K = 0.002.

c1 = A1 * (1 / ((2 * pi) ^ 0.5) * sig1);
k1 = 2 * (sig1 ^ 2);
c2 = A2 * (1 / ((2 * pi) ^ 0.5) * sig2);
k2 = 2 * (sig2 ^ 2);
z  = linspace(0, 1, 256);
```



a b
c d

FIGURE 3.10
(a) Image of the Mars moon Phobos. (b) Histogram. (c) Histogram-equalized image. (d) Histogram of (c). (Original image courtesy of NASA.)

```
p = k + c1 * exp(-((z - m1) .^ 2) ./ k1) + ...
    c2 * exp(-((z - m2) .^ 2) ./ k2);
p = p ./ sum(p(:));
```

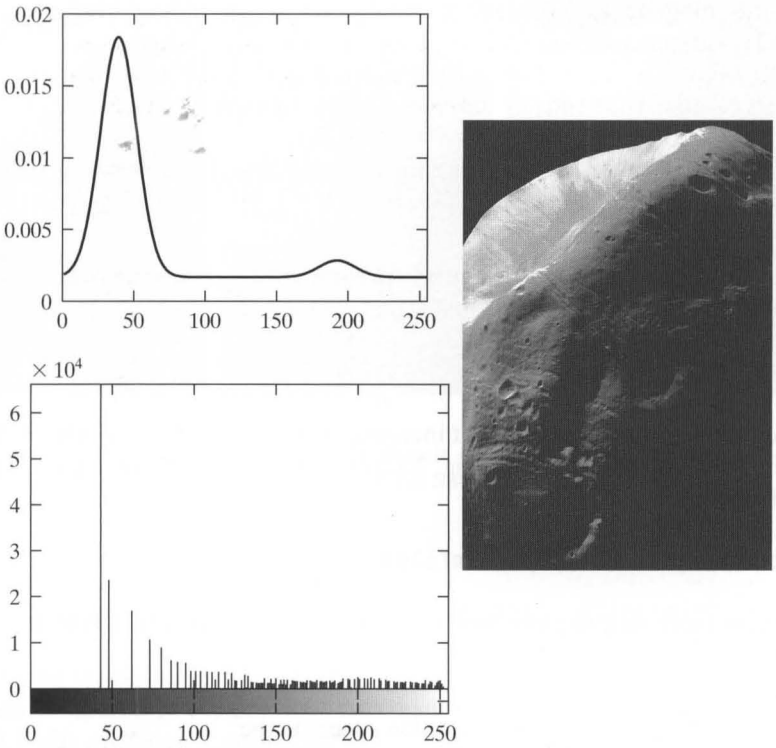
The following interactive function accepts inputs from a keyboard and plots the resulting Gaussian function. Refer to Section 2.10.6 for an explanation of function `input`. Note how the limits of the plots are set.

`manualhist`

```
function p = manualhist
%MANUALHIST Generates a two-mode histogram interactively.
%   P = MANUALHIST generates a two-mode histogram using function
%   TWOMODEGAUSS(m1, sig1, m2, sig2, A1, A2, k). m1 and m2 are the
%   means of the two modes and must be in the range [0,1]. SIG1 and
%   SIG2 are the standard deviations of the two modes. A1 and A2 are
%   amplitude values, and k is an offset value that raises the floor
```

```
%   of the the histogram. The number of elements in the histogram
%   vector P is 256 and sum(P) is normalized to 1. MANUALHIST
%   repeatedly prompts for the parameters and plots the resulting
%   histogram until the user types an 'x' to quit, and then it
%   returns the last histogram computed.
%
%   A good set of starting values is: (0.15, 0.05, 0.75, 0.05, 1,
%   0.07, 0.002).

% Initialize.
repeats = true;
quitnow = 'x';

% Compute a default histogram in case the user quits before
% estimating at least one histogram.
p = twomodegauss(0.15, 0.05, 0.75, 0.05, 1, 0.07, 0.002);

% Cycle until an x is input.
while repeats
   s = input('Enter m1, sig1, m2, sig2, A1, A2, k OR x to quit:',...
      's');
   if strcmp(s, quitnow)
      break
   end

   % Convert the input string to a vector of numerical values and
   % verify the number of inputs.
   v = str2num(s);
   if numel(v) ~= 7
      disp('Incorrect number of inputs.')
      continue
   end

   p = twomodegauss(v(1), v(2), v(3), v(4), v(5), v(6), v(7));
   % Start a new figure and scale the axes. Specifying only xlim
   % leaves ylim on auto.
   figure, plot(p)
   xlim([0 255])
end
```

Because the problem with histogram equalization in this example is due primarily to a large concentration of pixels in the original image with levels near 0, a reasonable approach is to modify the histogram of that image so that it does not have this property. Figure 3.11(a) shows a plot of a function (obtained with program `manualhist`) that preserves the general shape of the original histogram, but has a smoother transition of levels in the dark region of the intensity scale. The output of the program, `p`, consists of 256 equally spaced points from this function and is the desired specified histogram. An image with the specified histogram was generated using the command



a
c b

FIGURE 3.11
(a) Specified histogram.
(b) Result of enhancement by histogram matching.
(c) Histogram of (b).

```
>> g = histeq(f, p);
```

Figure 3.11(b) shows the result. The improvement over the histogram-equalized result in Fig. 3.10(c) is evident. Note that the specified histogram represents a rather modest change from the original histogram. This is all that was required to obtain a significant improvement in enhancement. The histogram of Fig. 3.11(b) is shown in Fig. 3.11(c). The most distinguishing feature of this histogram is how its low end has been moved closer to a lighter region of the gray scale, and thus closer to the specified shape. Note, however, that the shift to the right was not as extreme as the shift in the histogram in Fig. 3.10(d), which corresponds to the poorly enhanced image of Fig. 3.10(c). ■

3.3.4 Function `adapthisteq`

This toolbox function performs so-called *contrast-limited adaptive histogram equalization* (CLAHE). Unlike the methods discussed in the previous two sections, which operate on an entire image, this approach consists of processing small regions of the image (called *tiles)* using histogram specification for each tile individually. Neighboring tiles are then combined using bilinear interpolation to eliminate artificially induced boundaries. The contrast, especially in

See Section 6.6 regarding interpolation.

areas of homogeneous intensity, can be limited to avoid amplifying noise. The syntax for `adapthisteq` is




```
g = adapthisteq(f, param1, val1, param2, val2, ...)
```

where `f` is the input image, `g` is the output image, and the `param/val` pairs are as listed in Table 3.2.

EXAMPLE 3.7: Using function `adapthisteq`.

■ Figure 3.12(a) is the same as Fig. 3.10(a) and Fig. 3.12(b) is the result of using all the default settings in function `adapthisteq`:

```
>> g1 = adapthisteq(f);
```

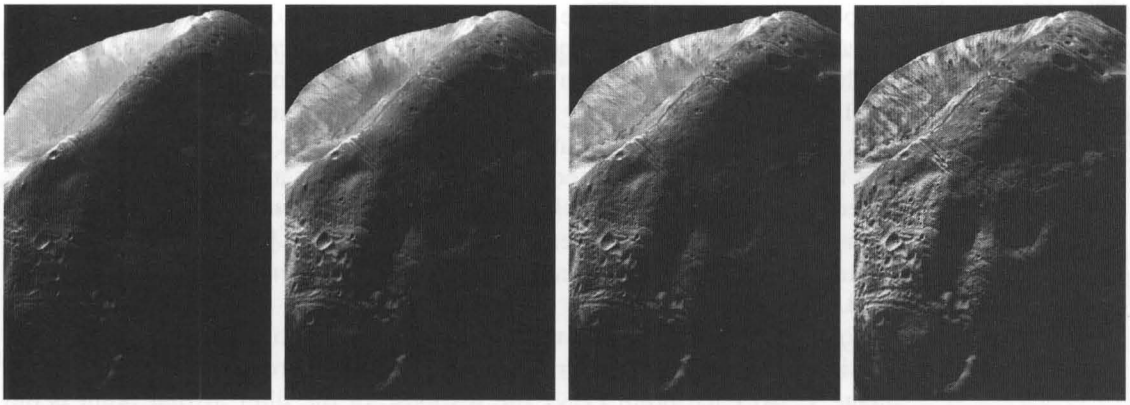
Although this result shows a slight increase in detail, significant portions of the image still are in the shadows. Fig. 3.12(c) shows the result of increasing the size of the tiles to [25 25]:

```
>> g2 = adapthisteq(f, 'NumTiles', [25 25]);
```

Sharpness increased slightly, but no new details are visible. Using the command

TABLE 3.2 Parameters and corresponding values for use in function `adapthisteq`.

Parameter	Value
`'NumTiles'`	Two-element vector of positive integers specifying the number of tiles by row and column, `[r c]`. Both `r` and `c` must be at least 2. The total number of tiles is equal to `r*c`. The default is `[8 8]`.
`'ClipLimit'`	Scalar in the range [0 1] that specifies a contrast enhancement limit. Higher numbers result in more contrast. The default is 0.01.
`'NBins'`	Positive integer scalar specifying the number of bins for the histogram used in building a contrast enhancing transformation. Higher values result in greater dynamic range at the cost of slower processing speed. The default is 256.
`'Range'`	A string specifying the range of the output image data: `'original'` — Range is limited to the range of the original image, `[min(f(:)) max(f(:))]`. `'full'` — Full range of the output image class is used. For example, for `uint8` data, range is [0 255]. This is the default.
`'Distribution'`	A string specifying the desired histogram shape for the image tiles: `'uniform'` — Flat histogram (this is the default). `'rayleigh'` — Bell-shaped histogram. `'exponential'` — Curved histogram. (See Section 5.2.2 for the equations for these distributions.
`'Alpha'`	Nonnegative scalar applicable to the Rayleigh and exponential distributions. The default value is 0.4.



a b c d

FIGURE 3.12 (a) Same as Fig. 3.10(a). (b) Result of using function `adapthisteq` with the default values. (c) Result of using this function with parameter `NumTiles` set to `[25 25]`. Result of using this number of tiles and `ClipLimit` = `0.05`.

```
>> g3 = adapthisteq(f, 'NumTiles', [25 25], 'ClipLimit', 0.05);
```

yielded the result in Fig. 3.12(d). The enhancement in detail in this image is significant compared to the previous two results. In fact, comparing Figs. 3.12(d) and 3.11(b) provides a good example of the advantage that local enhancement can have over global enhancement methods. Generally, the price paid is additional function complexity. ■

3.4 Spatial Filtering

As mentioned in Section 3.1 and illustrated in Fig. 3.1, neighborhood processing consists of (1) selecting a center point, (x, y); (2) performing an operation that involves only the pixels in a predefined neighborhood about (x, y); (3) letting the result of that operation be the "response" of the process at *that* point; and (4) repeating the process for every point in the image. The process of moving the center point creates new neighborhoods, one for each pixel in the input image. The two principal terms used to identify this operation are *neighborhood processing* and *spatial filtering*, with the second term being more prevalent. As explained in the following section, if the computations performed on the pixels of the neighborhoods are linear, the operation is called *linear spatial filtering* (the term *spatial convolution* also used); otherwise it is called *nonlinear spatial filtering*.

3.4.1 Linear Spatial Filtering

The concept of *linear filtering* has its roots in the use of the Fourier transform for signal processing in the frequency domain, a topic discussed in detail in Chapter 4. In the present chapter, we are interested in filtering operations that

are performed directly on the pixels of an image. Use of the term *linear spatial filtering* differentiates this type of process from *frequency domain filtering*.

The linear operations of interest in this chapter consist of multiplying each pixel in the neighborhood by a corresponding coefficient and summing the results to obtain the response at each point (x, y). If the neighborhood is of size $m \times n$, mn coefficients are required. The coefficients are arranged as a matrix, called a *filter*, *mask*, *filter mask*, *kernel*, *template*, or *window*, with the first three terms being the most prevalent. For reasons that will become obvious shortly, the terms *convolution filter*, *convolution mask*, or *convolution kernel*, also are used.

Figure 3.13 illustrates the mechanics of linear spatial filtering. The process consists of moving the center of the filter mask, w, from point to point in an image, f. At each point (x, y), the response of the filter at that point is the sum of products of the filter coefficients and the corresponding neighborhood pixels in the area spanned by the filter mask. For a mask of size $m \times n$, we assume typically that $m = 2a + 1$ and $n = 2b + 1$ where a and b are nonnegative integers. All this says is that our principal focus is on masks of odd sizes, with the smallest meaningful size being 3×3. Although it certainly is not a requirement, working with odd-size masks is more intuitive because they have an unambiguous center point.

There are two closely related concepts that must be understood clearly when performing linear spatial filtering. One is *correlation*; the other is *convolution*. Correlation is the process of passing the mask w by the image array f in the manner described in Fig. 3.13. Mechanically, convolution is the same process, except that w is rotated by 180° prior to passing it by f. These two concepts are best explained by some examples.

Figure 3.14(a) shows a one-dimensional function, f, and a mask, w. The origin of f is assumed to be its leftmost point. To perform the correlation of the two functions, we move w so that its rightmost point coincides with the origin of f, as Fig. 3.14(b) shows. Note that there are points between the two functions that do not overlap. The most common way to handle this problem is to pad f with as many 0s as are necessary to guarantee that there will always be corresponding points for the full excursion of w past f. This situation is illustrated in Fig. 3.14(c).

We are now ready to perform the correlation. The first value of correlation is the sum of products of the two functions in the position shown in Fig. 3.14(c). The sum of products is 0 in this case. Next, we move w one location to the right and repeat the process [Fig. 3.14(d)]. The sum of products again is 0. After four shifts [Fig. 3.14(e)], we encounter the first nonzero value of the correlation, which is (2)(1) = 2. If we proceed in this manner until w moves completely past f [the ending geometry is shown in Fig. 3.14(f)] we would get the result in Fig. 3.14(g). This set of values is the correlation of w and f. If we had padded w, aligned the rightmost element of f with the leftmost element of the padded w, and performed correlation in the manner just explained, the result would have been different (rotated by 180°), so order of the functions matters in correlation.

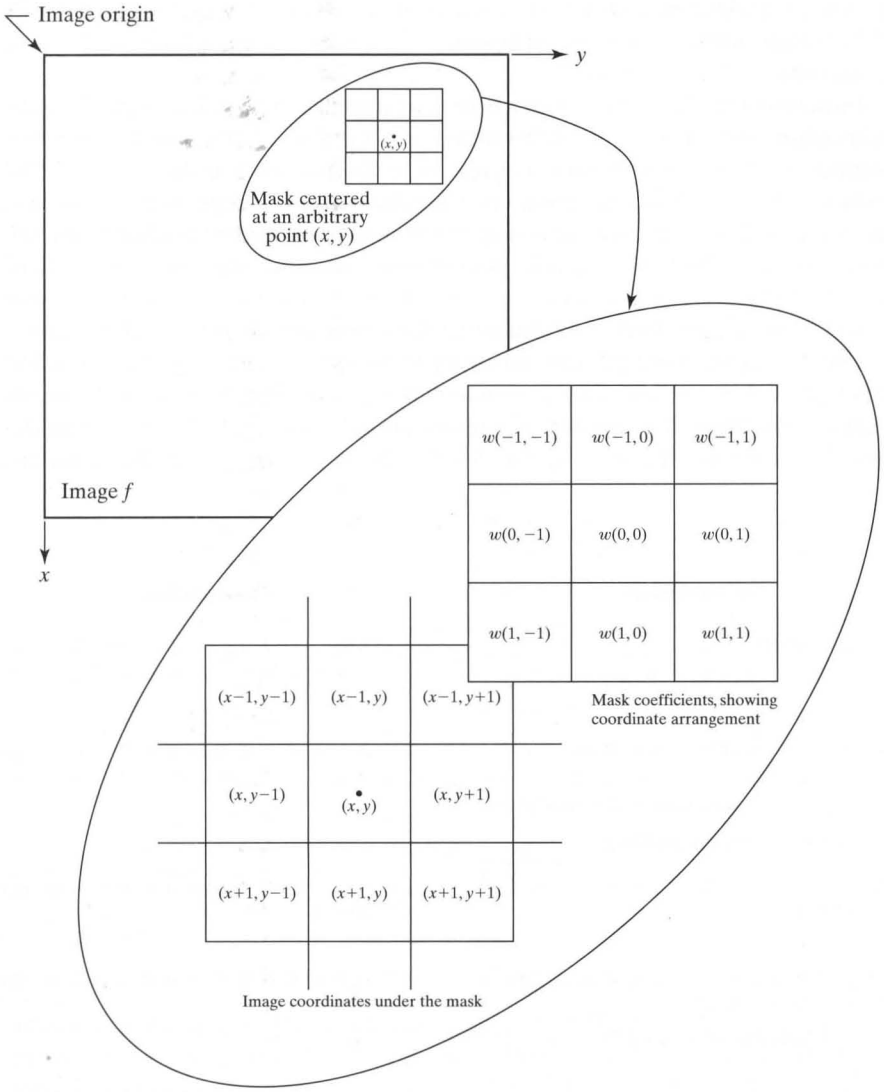


FIGURE 3.13
The mechanics of linear spatial filtering. The magnified drawing shows a 3×3 filter mask and the corresponding image neighborhood directly under it. The image neighborhood is shown displaced out from under the mask for ease of readability.

The label `'full'` in the correlation in Fig. 3.14(g) is a flag (to be discussed later) used by the toolbox to indicate correlation using a padded image and computed in the manner just described. The toolbox provides another option, denoted by `'same'` [Fig. 3.14(h)] that produces a correlation that is of the same size as f. This computation also uses zero padding, but the starting position is with the center point of the mask (the point labeled 3 in w) aligned with the origin of f. The last computation is with the center point of the mask aligned with the last point in f.

To perform convolution we rotate w by 180° and place its rightmost point at the origin of f, as Fig. 3.14(j) shows. We then repeat the sliding/computing

process employed in correlation, as illustrated in Figs. 3.14(k) through (n). The `'full'` and `'same'` convolution results are shown in Figs. 3.14(o) and (p), respectively.

Function f in Fig. 3.14 is a discrete unit impulse that is 1 at a point and 0 everywhere else. It is evident from the result in Figs. 3.14(o) or (p) that convolution with an impulse just "copies" w at the location of the impulse. This copying property (called *sifting*) is a fundamental concept in linear system theory, and it is the reason why one of the functions is always rotated by 180° in convolution. Note that, unlike correlation, swapping the order of the functions yields the same convolution result. If the function being shifted is symmetric, it is evident that convolution and correlation yield the same result.

The preceding concepts extend easily to images, as Fig. 3.15 illustrates. The origin is at the top, left corner of image $f(x, y)$ (see Fig. 2.1). To perform correlation, we place the bottom, rightmost point of $w(x, y)$ so that it coincides with the origin of $f(x, y)$ as in Fig. 3.15(c). Note the use of 0 padding for the

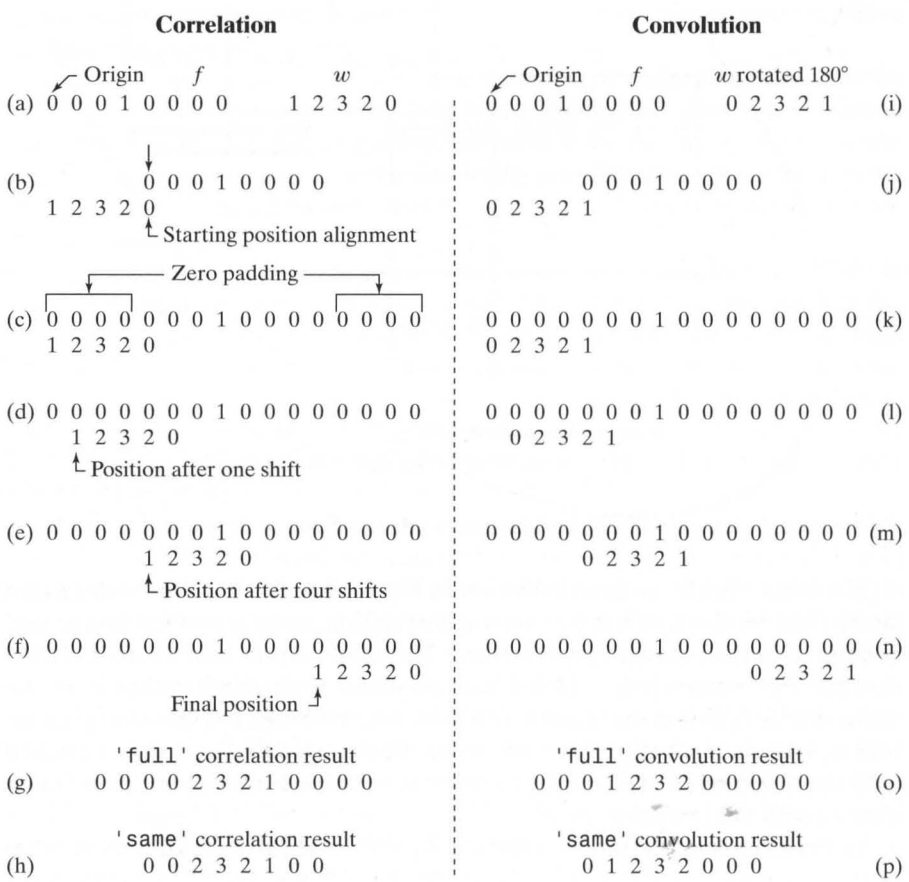


FIGURE 3.14 Illustration of one-dimensional correlation and convolution.

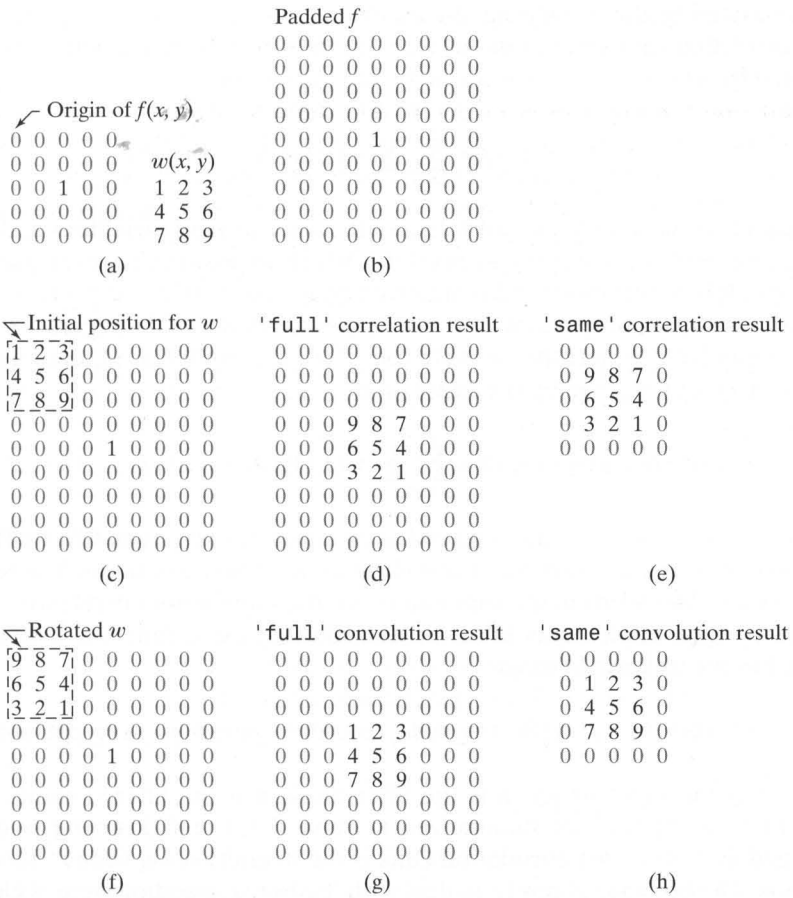


FIGURE 3.15 Illustration of two-dimensional correlation and convolution. The 0s are shown in gray to simplify viewing.

reasons mentioned in the discussion of Fig. 3.14. To perform correlation, we move $w(x, y)$ in all possible locations so that at least one of its pixels overlaps a pixel in the original image $f(x, y)$. This `'full'` correlation is shown in Fig. 3.15(d). To obtain the `'same'` correlation in Fig. 3.15(e), we require that all excursions of $w(x, y)$ be such that its center pixel overlaps the original $f(x, y)$. For convolution, we rotate $w(x, y)$ by 180° and proceed in the same manner as in correlation [see Figs. 3.15(f) through (h)]. As in the one-dimensional example discussed earlier, convolution yields the same result independently of the order of the functions. In correlation the order does matter, a fact that is made clear in the toolbox by assuming that the filter mask is always the function that undergoes translation. Note also the important fact in Figs. 3.15(e) and (h) that the results of spatial correlation and convolution are rotated by 180° with respect to each other. This, of course, is expected because convolution is nothing more than correlation with a rotated filter mask.

Summarizing the preceding discussion in equation form, we have that the correlation of a filter mask $w(x, y)$ of size $m \times n$ with a function $f(x, y)$, denoted by $w(x, y) \star f(x, y)$, is given by the expression

$$w(x, y) \star f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x + s, y + t)$$

This equation is evaluated for all values of the *displacement* variables x and y so that all elements of w visit every pixel in f, which we assume has been padded appropriately. Constants a and b are given by $a = (m - 1)/2$ and $b = (n - 1)/2$. For notational convenience, we assume that m and n are odd integers.

In a similar manner, the convolution of $w(x, y)$ and $f(x, y)$, denoted by $w(x, y) \star f(x, y)$, is given by the expression

$$w(x, y) \star f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x - s, y - t)$$

where the minus signs on the right of the equation flip f (i.e., rotate it by 180°). Rotating and shifting f instead of w is done to simplify the notation. The result is the same.[†] The terms in the summation are the same as for correlation.

The toolbox implements linear spatial filtering using function `imfilter`, which has the following syntax:




```
g = imfilter(f, w, filtering_mode, boundary_options, size_options)
```

where `f` is the input image, `w` is the filter mask, `g` is the filtered result, and the other parameters are summarized in Table 3.3. The `filtering_mode` is specified as `'corr'` for correlation (this is the default) or as `'conv'` for convolution. The `boundary_options` deal with the border-padding issue, with the size of the border being determined by the size of the filter. These options are explained further in Example 3.8. The `size_options` are either `'same'` or `'full'`, as explained in Figs. 3.14 and 3.15.

The most common syntax for `imfilter` is

```
g = imfilter(f, w, 'replicate')
```

This syntax is used when implementing standard linear spatial filters in the toolbox. These filters, which are discussed in Section 3.5.1, are prerotated by 180°, so we can use the correlation default in `imfilter` (from the discussion of Fig. 3.15, we know that performing correlation with a rotated filter is the same as performing convolution with the original filter). If the filter is symmetric about its center, then both options produce the same result.

[†]Because convolution is commutative, we have that $w(x, y) \star f(x, y) = f(x, y) \star w(x, y)$. This is not true of correlation, as you can see, for example, by reversing the order of the two functions in Fig. 3.14(a).

Options	Description
Filtering Mode	
`'corr'`	Filtering is done using correlation (see Figs. 3.14 and 3.15). This is the default.
`'conv'`	Filtering is done using convolution (see Figs. 3.14 and 3.15).
Boundary Options	
`P`	The boundaries of the input image are extended by padding with a value, `P` (written without quotes). This is the default, with value 0.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'circular'`	The size of the image is extended by treating the image as one period a 2-D periodic function.
Size Options	
`'full'`	The output is of the same size as the extended (padded) image (see Figs. 3.14 and 3.15).
`'same'`	The output is of the same size as the input. This is achieved by limiting the excursions of the center of the filter mask to points contained in the original image (see Figs. 3.14 and 3.15). This is the default.

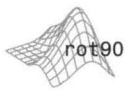
TABLE 3.3 Options for function `imfilter`.

When working with filters that are neither pre-rotated nor symmetric, and we wish to perform convolution, we have two options. One is to use the syntax

```
g = imfilter(f, w, 'conv', 'replicate')
```

The other approach is to use function `rot90(w, 2)` to rotate `w` by 180°, and then use `imfilter(f, w, 'replicate')`. The two steps can be combined into one:

```
g = imfilter(f, rot90(w, 2), 'replicate')
```



`rot90(w, k)` rotates `w` by `k*90` degrees, where `k` is an integer.

The result would be an image, `g`, that is of the same size as the input (i.e., the default is the `'same'` mode discussed earlier).

Each element of the filtered image is computed using floating-point arithmetic. However, `imfilter` converts the output image to the same class of the input. Therefore, if `f` is an integer array, then output elements that exceed the range of the integer type are truncated, and fractional values are rounded. If more precision is desired in the result, then `f` should be converted to floating point using functions `im2single`, `im2double`, or `tofloat` (see Section 2.7) before using `imfilter`.

EXAMPLE 3.8: Using function `imfilter`.

■ Figure 3.16(a) is a class `double` image, `f`, of size 512 × 512 pixels. Consider the 31 × 31 filter

```
>> w = ones(31);
```

which is proportional to an averaging filter. We did not divide the coefficients by $(31)^2$ to illustrate at the end of this example the scaling effects of using `imfilter` with an image of class `uint8`.

Convolving filter `w` with an image produces a blurred result. Because the filter is symmetric, we can use the correlation default in `imfilter`. Figure 3.16(b) shows the result of performing the following filtering operation:

```
>> gd = imfilter(f, w);
>> imshow(gd, [ ])
```

where we used the default boundary option, which pads the border of the image with 0s (black). As expected, the edges between black and white in the filtered image are blurred, but so are the edges between the light parts of the image and the boundary. The reason is that the padded border is black. We can deal with this difficulty by using the `'replicate'` option

```
>> gr = imfilter(f, w, 'replicate');
>> figure, imshow(gr, [ ])
```

As Fig. 3.16(c) shows, the borders of the filtered image now appear as expected. In this case, equivalent results are obtained with the `'symmetric'` option

```
>> gs = imfilter(f, w, 'symmetric');
>> figure, imshow(gs, [ ])
```

Figure 3.16(d) shows the result. However, using the `'circular'` option

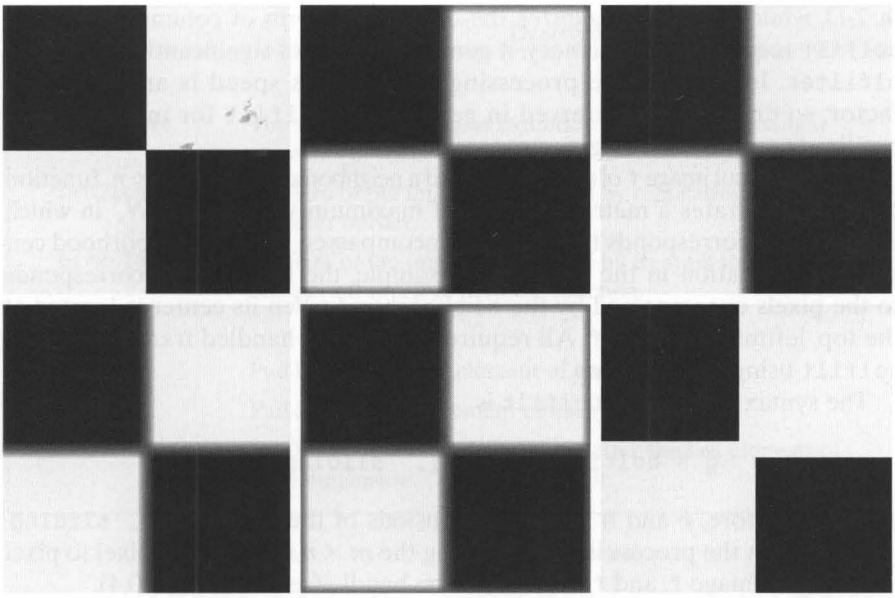
```
>> gc = imfilter(f, w, 'circular');
>> figure, imshow(gc, [ ])
```

produced the result in Fig. 3.16(e), which shows the same problem as with zero padding. This is as expected because use of periodicity makes the black parts of the image adjacent to the light areas.

Finally, we illustrate how the fact that `imfilter` produces a result that is of the same class as the input can lead to difficulties if not handled properly:

```
>> f8 = im2uint8(f);
>> g8r = imfilter(f8, w, 'replicate');
>> figure, imshow(g8r, [ ])
```

Figure 3.16(f) shows the result of these operations. Here, when the output was converted to the class of the input (`uint8`) by `imfilter`, clipping caused



a b c
d e f

FIGURE 3.16
(a) Original image. (b) Result of using `imfilter` with default zero padding. (c) Result with the `'replicate'` option. (d) Result with the `'symmetric'` option. (e) Result with the `'circular'` option. (f) Result of converting the original image to class `uint8` and then filtering with the `'replicate'` option. A filter of size 31×31 with all 1s was used throughout.

some data loss. The reason is that the coefficients of the mask did not sum to the range [0, 1], resulting in filtered values outside the [0, 255] range. Thus, to avoid this difficulty, we have the option of normalizing the coefficients so that their sum is in the range [0, 1] (in the present case we would divide the coefficients by $(31)^2$ so the sum would be 1), or inputting the data in `single` or `double` format. Note, however, that even if the second option were used, the data usually would have to be normalized to a valid image format at some point (e.g., for storage) anyway. Either approach is valid; the key point is that data ranges have to be kept in mind to avoid unexpected results. ■

3.4.2 Nonlinear Spatial Filtering

Nonlinear spatial filtering is based on neighborhood operations also, and the mechanics of sliding the center point of an $m \times n$ filter through an image are the same as discussed in the previous section. However, whereas linear spatial filtering is based on computing the sum of products (which is a linear operation), nonlinear spatial filtering is based, as the name implies, on nonlinear operations involving the pixels in the neighborhood encompassed by the filter. For example, letting the response at each center point be equal to the maximum pixel value in its neighborhood is a nonlinear filtering operation. Another basic difference is that the concept of a mask is not as prevalent in nonlinear processing. The idea of filtering carries over, but the "filter" should be visualized as a nonlinear function that operates on the pixels of a neighborhood, and whose response constitutes the result of the nonlinear operation.

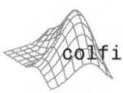
The toolbox provides two functions for performing general nonlinear filtering: `nlfilter` and `colfilt`. The former performs operations directly



in 2-D, while `colfilt` organizes the data in the form of columns. Although `colfilt` requires more memory, it generally executes significantly faster than `nlfilter`. In most image processing applications speed is an overriding factor, so `colfilt` is preferred in general over `nlfilt` for implementing nonlinear spatial filtering.

Given an input image `f` of size $M \times N$ and a neighborhood of size $m \times n$, function `colfilt` generates a matrix, call it `A`, of maximum size $mn \times MN$,[†] in which each column corresponds to the pixels encompassed by the neighborhood centered at a location in the image. For example, the first column corresponds to the pixels encompassed by the neighborhood when its center is located at the top, leftmost point in `f`. All required padding is handled transparently by `colfilt` using zero padding.

The syntax of function `colfilt` is




```
g = colfilt(f, [m n], 'sliding', fun)
```

where, as before, `m` and `n` are the dimensions of the filter region, `'sliding'` indicates that the process is one of sliding the $m \times n$ region from pixel to pixel in the input image `f`, and `fun` is a function handle (see Section 2.10.4).

Because of the way in which matrix `A` is organized, function `fun` must operate on each of the columns of `A` individually and return a row vector, `v`, whose kth element is the result of the operation performed by `fun` on the kth column of `A`. Because there can be up to MN columns in `A`, the maximum dimension of `v` is $1 \times MN$.

The linear filtering discussed in the previous section has provisions for padding to handle the border problems inherent in spatial filtering. When using `colfilt`, however, the input image must be padded explicitly before filtering. For this we use function `padarray`, which, for 2-D functions, has the syntax




```
fp = padarray(f, [r c], method, direction)
```

where `f` is the input image, `fp` is the padded image, `[r c]` gives the number of rows and columns by which to pad `f`, and `method` and `direction` are as explained in Table 3.4. For example, if `f = [1 2; 3 4]`, the command

```
>> fp = padarray(f, [3 2], 'replicate', 'post')
```

produces the result

```
fp =
     1     2     2     2
     3     4     4     4
     3     4     4     4
     3     4     4     4
     3     4     4     4
```

[†] `A` *always* has mn rows, but the number of columns can vary, depending on the size of the input. Size selection is managed automatically by `colfilt`.

Options	Description
Method	
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'circular'`	The size of the image is extended by treating the image as one period of a 2-D periodic function.
Direction	
`'pre'`	Pad before the first element of each dimension.
`'post'`	Pad after the last element of each dimension.
`'both'`	Pad before the first element and after the last element of each dimension. This is the default.

TABLE 3.4 Options for function `padarray`.

If `direction` is not included in the argument, the default is `'both'`. If `method` is not included, the default padding is with 0s.

■ As an illustration of function `colfilt`, we implement a nonlinear filter whose response at any point is the geometric mean of the intensity values of the pixels in the neighborhood centered at that point. The geometric mean in a neighborhood of size $m \times n$ is the product of the intensity values in the neighborhood raised to the power $1/mn$. First we implement the nonlinear filter function as an anonymous function handle (see Section 2.10.4):

EXAMPLE 3.9: Using function `colfilt` to implement a nonlinear spatial filter.

```
>> gmean = @(A) prod(A, 1)^1/size(A, 1));
```



If A is a vector, `prod(A)` returns the product of the elements. If A is a matrix, `prod(A)` treats the columns as vectors and returns a row vector of the products of each column. `prod(A, dim)` computes the product along the dimension of A specified by `dim`. See the `prod` help page for details on how this function behaves when A is a multidimensional array.

To reduce border effects, we pad the input image using, say, the `'replicate'` option in function `padarray`:

```
f = padarray(f, [m n], 'replicate');
```

Next, we call `colfilt`:

```
>> g = colfilt(f, [m n], 'sliding', @gmean);
```

There are several important points at play here. First, note that matrix `A` is automatically passed to the function handle `gmean` by `colfilt`. Second, as mentioned earlier, matrix `A` always has mn rows, but the number of columns is variable. Therefore `gmean` (or any other function handle passed by `colfilt`) has to be written in a manner that can handle a variable number of columns.

The filtering process in this case consists of computing the product of all pixels in the neighborhood and then raising the result to the power $1/mn$. For

any value of (x, y) the filtered result at that point is contained in the appropriate column in v. The key requirement is that the function operate on the columns of A, no matter how many there are, and return a row vector containing the result for all individual columns. Function colfilt then takes those results and rearranges them to produce the output image, g.

Finally, we remove the padding inserted earlier:

```
>> [M, N] = size(f);
>> g = g((1:M) + m, (1:N) + n);
```

so that g is of the same size as f. ■

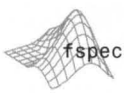
Some commonly used nonlinear filters can be implemented in terms of other MATLAB and toolbox functions such as imfilter and ordfilt2 (see Section 3.5.2). Function spfilt in Section 5.3, for example, implements the geometric mean filter in Example 3.9 in terms of imfilter and the MATLAB log and exp functions. When this is possible, performance usually is much faster, and memory usage is a fraction of the memory required by colfilt. However, colfilt remains the best choice for nonlinear filtering operations that do not have such alternate implementations.

3.5 Image Processing Toolbox Standard Spatial Filters

In this section we discuss linear and nonlinear spatial filters supported by the toolbox. Additional custom filter functions are implemented in Section 5.3.

3.5.1 Linear Spatial Filters

The toolbox supports a number of predefined 2-D linear spatial filters, obtained by using function fspecial, which generates a filter mask, w, using the syntax




```
w = fspecial('type', parameters)
```

where 'type' specifies the filter type, and parameters further define the specified filter. The spatial filters that fspecial can generate are summarized in Table 3.5, including applicable parameters for each filter.

EXAMPLE 3.10: Using function imfilter to implement a Laplacian filter.

■ We illustrate the use of fspecial and imfilter by enhancing an image with a Laplacian filter. The Laplacian of an image $f(x, y)$, denoted $\nabla^2 f(x, y)$, is defined as

$$\nabla^2 f(x, y) = \frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2}$$

We discuss digital approximations to first- and second-order derivatives in Section 11.1.3.

Commonly used digital approximations of the second derivatives are

$$\frac{\partial^2 f(x, y)}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y)$$

TABLE 3.5 Spatial filters supported by function `fspecial`. Several of the filters in this table are used for edge detection in Section 11.1.

Type	Syntax and Parameters
`'average'`	`fspecial('average', [r c])`. A rectangular averaging filter of size `r` × `c`. The default is 3 × 3. A single number instead of `[r c]` specifies a square filter.
`'disk'`	`fspecial('disk', r)`. A circular averaging filter (within a square of size `2r + 1`) with radius `r`. The default radius is 5.
`'gaussian'`	`fspecial('gaussian', [r c], sig)`. A Gaussian lowpass filter of size `r` × `c` and standard deviation `sig` (positive). The defaults are 3 × 3 and 0.5. A single number instead of `[r c]` specifies a square filter.
`'laplacian'`	`fspecial('laplacian', alpha)`. A 3 × 3 Laplacian filter whose shape is specified by `alpha`, a number in the range [0, 1]. The default value for `alpha` is 0.2.
`'log'`	`fspecial('log', [r c], sig)`. Laplacian of a Gaussian (LoG) filter of size `r` × `c` and standard deviation `sig` (positive). The defaults are 5 × 5 and 0.5. A single number instead of `[r c]` specifies a square filter.
`'motion'`	`fspecial('motion', len, theta)`. Outputs a filter that, when convolved with an image, approximates linear motion (of a camera with respect to the image) of `len` pixels. The direction of motion is `theta`, measured in degrees, counterclockwise from the horizontal. The defaults are 9 and 0, which represents a motion of 9 pixels in the horizontal direction.
`'prewitt'`	`fspecial('prewitt')`. Outputs a 3 × 3 Prewitt filter, `wv`, that approximates a vertical gradient. A filter mask for the horizontal gradient is obtained by transposing the result: `wh = wv'`.
`'sobel'`	`fspecial('sobel')`. Outputs a 3 × 3 Sobel filter, `sv`, that approximates a vertical gradient. A filter for the horizontal gradient is obtained by transposing the result: `sh = sv'`.
`'unsharp'`	`fspecial('unsharp', alpha)`. Outputs a 3 × 3 unsharp filter; `alpha` controls the shape; it must be in the range [0, 1]; the default is 0.2.

See Sections 7.6.1 and 11.1.3 regarding the gradient.

and

$$\frac{\partial^2 f(x,y)}{\partial y^2} = f(x,y+1) + f(x,y-1) - 2f(x,y)$$

so that

$$\nabla^2 f(x,y) = [f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1)] - 4f(x,y)$$

This expression can be implemented at all points in an image by convolving the image with the following spatial mask:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

An alternate definition of the digital second derivatives takes into account diagonal elements, and can be implemented using the mask

$$\begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix}$$

Both derivatives sometimes are defined with the signs opposite to those shown here, resulting in masks that are the negatives of the preceding two masks.

Enhancement using the Laplacian is based on the equation

$$g(x,y) = f(x,y) + c\left[\nabla^2 f(x,y)\right]$$

where $f(x,y)$ is the input image, $g(x,y)$ is the enhanced image, and c is 1 if the center coefficient of the mask is positive, or -1 if it is negative (Gonzalez and Woods [2008]). Because the Laplacian is a derivative operator, it sharpens the image but drives constant areas to zero. Adding the original image back restores the gray-level tonality.

Function `fspecial('laplacian', alpha)` implements a more general Laplacian mask:

$$\begin{matrix} \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \\ \frac{1-\alpha}{1+\alpha} & \frac{-4}{1+\alpha} & \frac{1-\alpha}{1+\alpha} \\ \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \end{matrix}$$

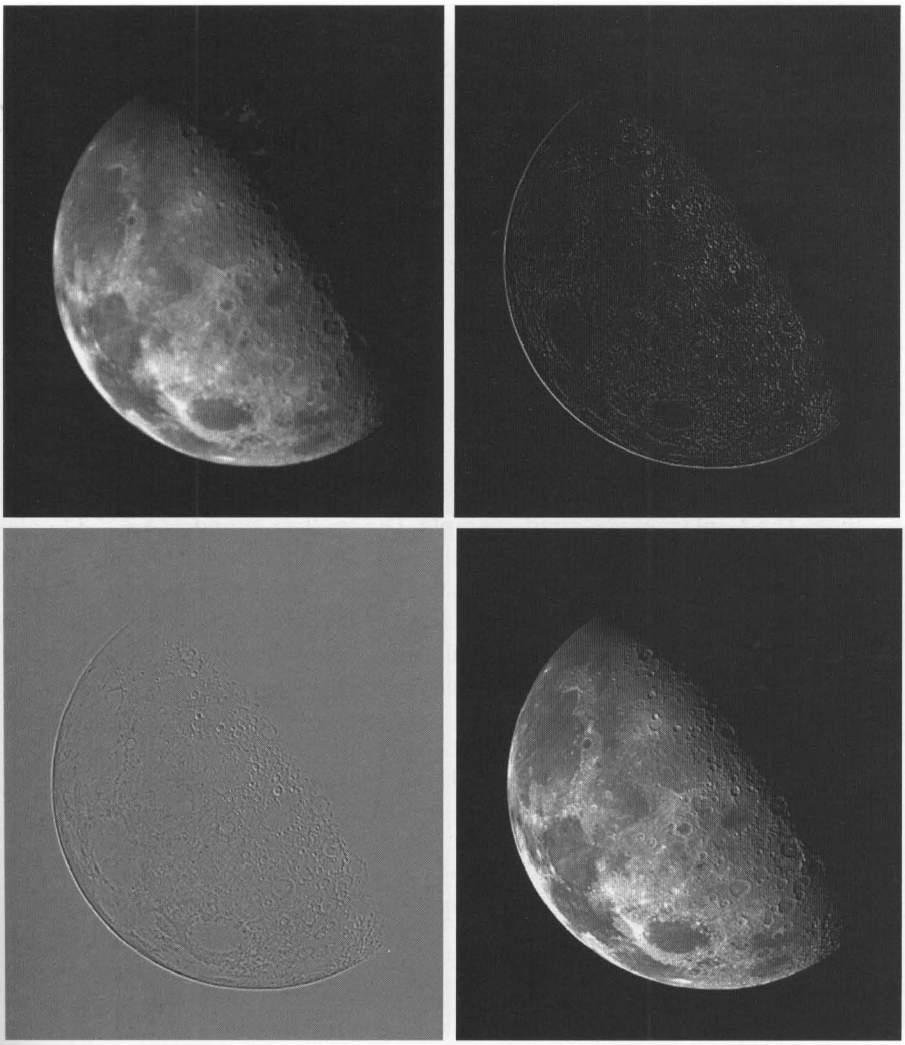
which allows fine tuning of enhancement results. However, the predominant use of the Laplacian is based on the two masks just discussed.

We now proceed to enhance the image in Fig. 3.17(a) using the Laplacian. This image is a mildly blurred image of the North Pole of the moon. Enhancement in this case consists of sharpening the image, while preserving as much of its gray tonality as possible. First, we generate and display the Laplacian filter:

```
>> w = fspecial('laplacian', 0)
w =
    0.0000    1.0000    0.0000
    1.0000   -4.0000    1.0000
    0.0000    1.0000    0.0000
```

Note that the filter is of class `double`, and that its shape with `alpha = 0` is the Laplacian filter discussed previously. We could just as easily have specified this shape manually as

```
>> w = [0 1 0; 1 -4, 1; 0 1 0];
```



a b
c d

FIGURE 3.17
(a) Image of the North Pole of the moon. (b) Laplacian filtered image, using `uint8` format. (Because `uint8` is an unsigned type, negative values in the output were clipped to 0.) (c) Laplacian filtered image obtained using floating point. (d) Enhanced result, obtained by subtracting (c) from (a). (Original image courtesy of NASA.)

Next we apply `w` to the input image, `f` [Fig 3.17(a)], which is of class `uint8`:

```
>> g1 = imfilter(f, w, 'replicate');
>> imshow(g1, [ ])
```

Figure 3.17(b) shows the resulting image. This result looks reasonable, but has a problem: all its pixels are positive. Because of the negative center filter coefficient, we know that we can expect in general to have a Laplacian image with positive and negative values. However, `f` in this case is of class `uint8` and, as discussed in the previous section, `imfilter` gives an output that is of the same class as the input image, so negative values are truncated. We get around this difficulty by converting `f` to floating point before filtering it:

```
>> f2 = tofloat(f);
>> g2 = imfilter(f2, w, 'replicate');
>> imshow(g2, [ ])
```

The result, shown in Fig. 3.17(c), is typical of the appearance of a Laplacian image.

Finally, we restore the gray tones lost by using the Laplacian by subtracting (because the center coefficient is negative) the Laplacian image from the original image:

```
>> g = f2 - g2;
>> imshow(g);
```

The result, shown in Fig. 3.17(d), is sharper than the original image. ■

EXAMPLE 3.11: Manually specifying filters and comparing enhancement techniques.

■ Enhancement problems often require filters beyond those available in the toolbox. The Laplacian is a good example. The toolbox supports a 3×3 Laplacian filter with a -4 in the center. Usually, sharper enhancement is obtained by using the 3×3 Laplacian filter that has a -8 in the center and is surrounded by 1s, as discussed earlier. The purpose of this example is to implement this filter manually, and also to compare the results obtained by using the two Laplacian formulations. The sequence of commands is as follows:

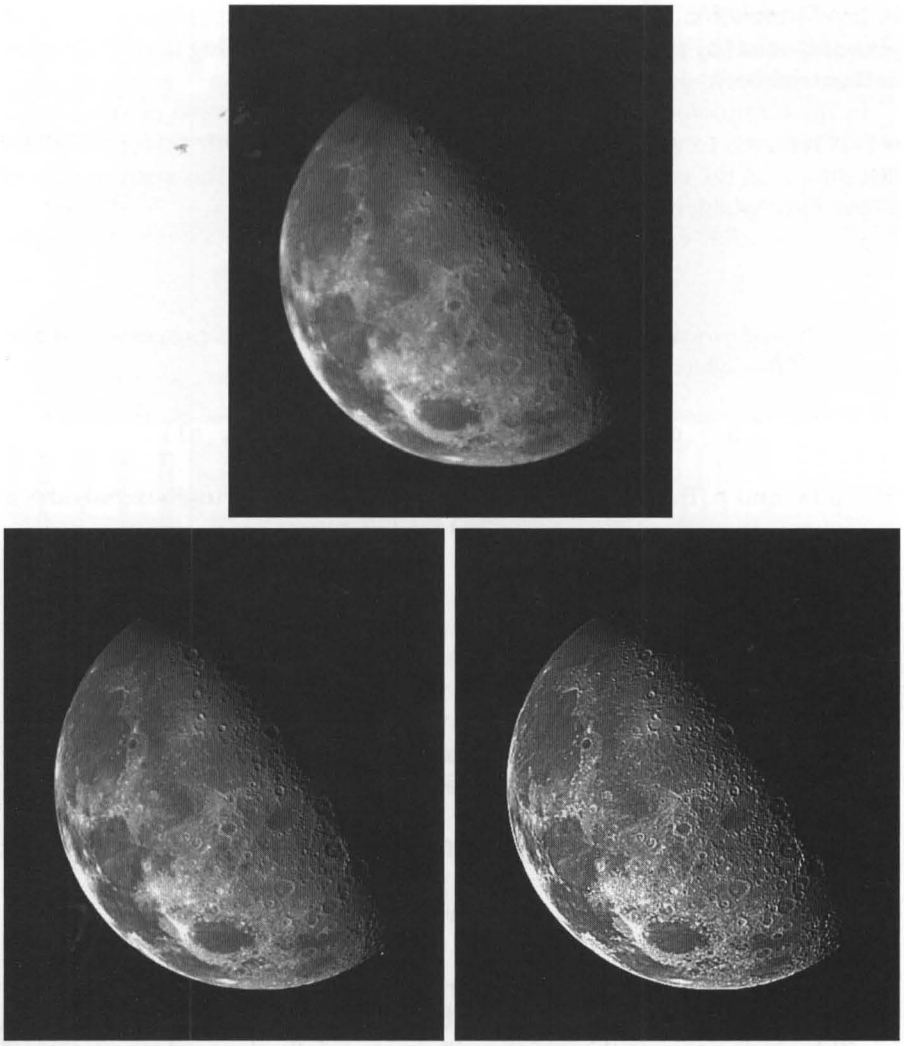
```
>> f = imread('Fig0317(a).tif');
>> w4 = fspecial('laplacian', 0); %Same as w in Example 3.10.
>> w8 = [1 1 1; 1 -8 1; 1 1 1];
>> f = tofloat(f);
>> g4 = f - imfilter(f, w4, 'replicate');
>> g8 = f - imfilter(f, w8, 'replicate');
>> imshow(f)
>> figure, imshow(g4)
>> figure, imshow(g8)
```

Figure 3.18(a) shows the original moon image again for easy comparison. Fig. 3.18(b) is `g4`, which is the same as Fig. 3.17(d), and Fig. 3.18(c) shows `g8`. As expected, this result is significantly sharper than Fig. 3.18(b). ■

3.5.2 Nonlinear Spatial Filters

Function `ordfilt2`, computes *order-statistic filters* (also called *rank filters*). These are nonlinear spatial filters whose response is based on ordering (ranking) the pixels contained in an image neighborhood and then replacing the value of the center pixel in the neighborhood with the value determined by the ranking result. Attention is focused in this section on nonlinear filters generated by `ordfilt2`. Several additional custom nonlinear filter functions are developed and implemented in Section 5.3.

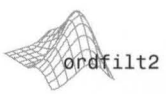
The syntax for function `ordfilt2` is



a
b c

FIGURE 3.18
(a) Image of the North Pole of the moon. (b) Image enhanced using the Laplacian filter `'laplacian'`, which has a −4 in the center. (c) Image enhanced using a Laplacian filter with a −8 in the center.

```
g = ordfilt2(f, order, domain)
```



This function creates the output image `g` by replacing each element of `f` by the `order`-th element in the sorted set of neighbors specified by the nonzero elements in `domain`. Here, `domain` is an $m \times n$ matrix of 1s and 0s that specify the pixel locations in the neighborhood that are to be used in the computation. In this sense, `domain` acts like a `logical` mask. The pixels in the neighborhood that correspond to 0 in the `domain` matrix are not used in the computation. For example, to implement a *min filter* (order 1) of size $m \times n$ we use the syntax

```
g = ordfilt2(f, 1, ones(m, n))
```

In this formulation the 1 denotes the 1st sample in the ordered set of mn samples, and `ones(m, n)` creates an $m \times n$ matrix of 1s, indicating that all samples in the neighborhood are to be used in the computation.

In the terminology of statistics, a *min filter* (the first sample of an ordered set) is referred to as the 0th percentile. Similarly, the 100th percentile is the last sample in the ordered set, which is the mn-th sample. This corresponds to a *max filter*, which is implemented using the syntax

As discussed in Chapter 10, another way to implement max and min filters is to use morphological erosion and dilation.

```
g = ordfilt2(f, m*n, ones(m, n))
```

The best-known order-statistic filter in digital image processing is the *median*[†] *filter*, which corresponds to the 50th percentile:

```
g = ordfilt2(f, (m*n + 1)/2, ones(m, n))
```

for odd `m` and `n`. Because of its practical importance, the toolbox provides a specialized implementation of the 2-D median filter:




```
g = medfilt2(f, [m n], padopt)
```

where the tuple `[m n]` defines a neighborhood of size $m \times n$ over which the median is computed, and `padopt` specifies one of three possible border padding options: `'zeros'` (the default), `'symmetric'` in which `f` is extended symmetrically by mirror-reflecting it across its border, and `'indexed'`, in which `f` is padded with 1s if it is of class `double` and with 0s otherwise. The default,

```
g = medfilt2(f)
```

uses a 3×3 neighborhood and pads the border of the input with 0s.

EXAMPLE 3.12: Median filtering with function `medfilt2`.

■ Median filtering is a useful tool for reducing salt-and-pepper noise in an image. Although we discuss noise reduction in much more detail in Chapter 5, it will be instructive at this point to illustrate briefly the implementation of median filtering.

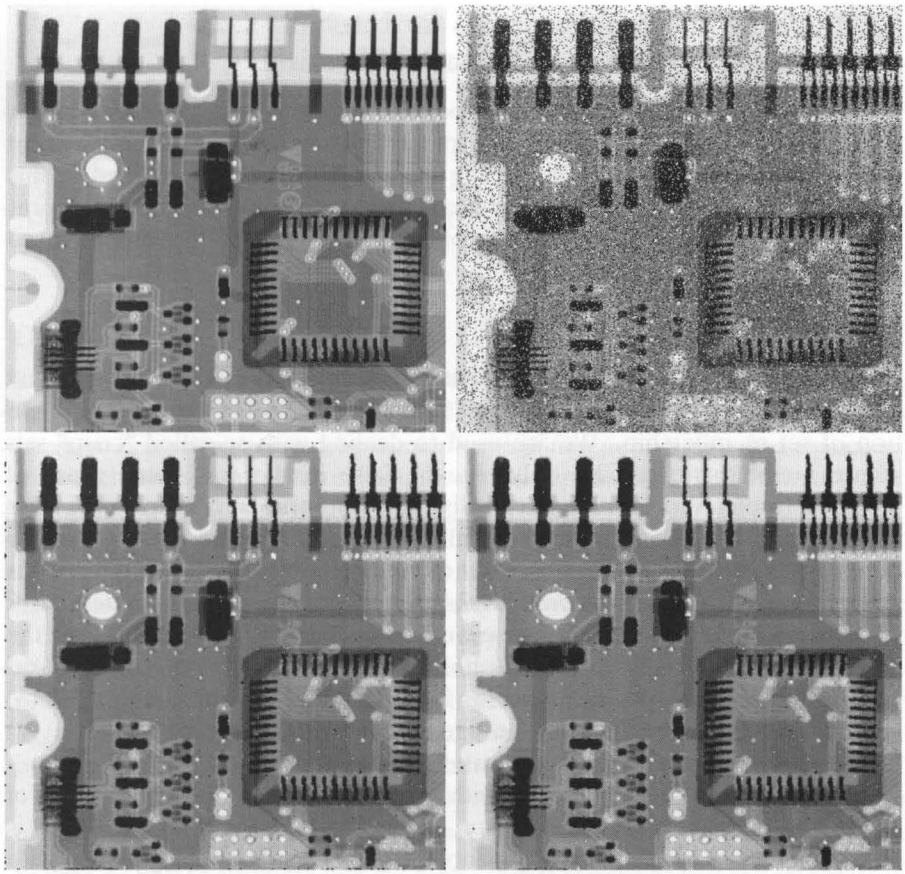
The image in Fig. 3.19(a) is an X-ray image, `f`, of an industrial circuit board taken during automated inspection of the board. Figure 3.19(b) is the same image corrupted by salt-and-pepper noise in which both the black and white points have a probability of occurrence of 0.2. This image was generated using function `imnoise`, which is discussed in Section 5.2.1:




```
>> fn = imnoise(f, 'salt & pepper', 0.2);
```

[†] Recall that the median, ξ, of a set of values is such that half the values in the set are less than or equal to ξ and half are greater than or equal to ξ. Although the discussion in this section is focused on images, MATLAB provides a general function, `median`, for computing median values of arrays of arbitrary dimension. See the `median` help page for details regarding this function.





a b
c d

FIGURE 3.19 Median filtering: (a) X-ray image. (b) Image corrupted by salt-and-pepper noise. (c) Result of median filtering with `medfilt2` using the default settings. (d) Result of median filtering using the `'symmetric'` option. Note the improvement in border behavior between (d) and (c). (Original image courtesy of Lixi, Inc.)

Figure 3.19(c) is the result of median filtering this noisy image, using the statement:

```
>> gm = medfilt2(fn);
```

Considering the level of noise in Fig. 3.19(b), median filtering using the default settings did a good job of noise reduction. Note, however, the black specks around the border. These were caused by the black points surrounding the image (recall that the default pads the border with 0s). This type of effect can be reduced by using the `'symmetric'` option:

```
>> gms = medfilt2(fn, 'symmetric');
```

The result, shown in Fig. 3.19(d), is close to the result in Fig. 3.19(c), except that the black border effect is not as pronounced. ■

3.6 Using Fuzzy Techniques for Intensity Transformations and Spatial Filtering

We conclude this chapter with an introduction to fuzzy sets and their application to intensity transformations and spatial filtering. We also develop a set of custom M-functions for implementing the fuzzy methods developed in this section. As you will see shortly, fuzzy sets provide a framework for incorporating human knowledge in the solution of problems whose formulation is based on imprecise concepts.

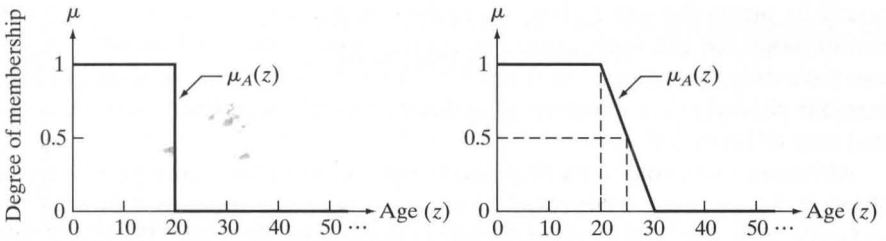
3.6.1 Background

A *set* is a collection of objects (*elements*) and *set theory* consists of tools that deal with operations on and among sets. Central to set theory is the notion of set membership. We are used to dealing with so-called "crisp" sets, whose membership can be only true or false in the traditional sense of bivalued Boolean logic, with 1 typically indicating true and 0 indicating false. For example, let Z denote the set of all people, and suppose that we want to define a subset, A, of Z, called the "set of young people." In order to form this subset, we need to define a *membership function* that assigns a value of 1 or 0 to every element, z, of Z. Because we are dealing with a bivalued logic, the membership function defines a threshold at or below which a person is considered young, and above which a person is considered not young. Figure 3.20(a) summarizes this concept using an age threshold of 20 years, where $\mu_A(z)$ denotes the membership function just discussed.

We see immediately a difficulty with this formulation: A person 20 years of age is considered young, but a person whose age is 20 years and 1 second is not a member of the set of young people. This is a fundamental problem with crisp sets that limits their use in many practical applications. What we need is more flexibility in what we mean by "young;" that is, a *gradual* transition from young to not young. Figure 3.20(b) shows one possibility. The essential feature of this function is that it is infinite-valued, thus allowing a continuous transition between young and not young. This makes it possible to have *degrees* of "youngness." We can make statements now such as a person being young (upper flat end of the curve), relatively young (toward the beginning of the ramp), 50% young (in the middle of the ramp), not so young (toward the end of the ramp), and so on (note that decreasing the slope of the curve in Fig. 3.20(b) introduces more vagueness in what we mean by "young"). These types of vague (*fuzzy*) statements are more consistent with what we humans use when talking imprecisely about age. Thus, we may interpret infinite-valued membership functions as being the foundation of a *fuzzy logic*, and the sets generated using them may be viewed as *fuzzy sets*.

3.6.2 Introduction to Fuzzy Sets

Fuzzy set theory was introduced by L. A. Zadeh (Zadeh [1965]) more than four decades ago. As the following discussion shows, fuzzy sets provide a formalism for dealing with imprecise information.



a b

FIGURE 3.20
Membership functions of (a) a crisp set, and (b) a fuzzy set.

Definitions

Let Z be a set of elements (objects), with a generic element of Z denoted by z; that is, $Z = \{z\}$. Set Z often is referred to as the *universe of discourse*. A *fuzzy set* A in Z is characterized by a *membership function*, $\mu_A(z)$, that associates with each element of Z a real number in the interval [0, 1]. For a particular element z_0 from Z, the value of $\mu_A(z_0)$ represents the *degree of membership* of z_0 in A.

The term *grade of membership* is used also to denote what we have defined as the degree of membership.

The concept of "belongs to," so familiar in ordinary (crisp) sets, does not have the same meaning in fuzzy set theory. With ordinary sets we say that an element either belongs or does not belong to a set. With fuzzy sets we say that all z's for which $\mu_A(z) = 1$ are *full* members of the set A, all z's for which $\mu_A(z)$ is between 0 and 1 have *partial* membership in the set, and all z's for which $\mu_A(z) = 0$ have *zero* degree of membership in the set (which, for all practical purposes, means that they are not members of the set).

For example, in Fig. 3.20(b) $\mu_A(25) = 0.5$, indicating that a person 25 years old has a 0.5 grade membership in the set of young people. Similarly two people of ages 15 and 35 have 1.0 and 0.0 grade memberships in this set, respectively. Therefore, a fuzzy set, A, is an *ordered pair* consisting of values of z and a membership function that assigns a grade of membership in A to each z. That is,

$$A = \{z, \mu_A(z) \mid z \in Z\}$$

When z is continuous, A can have an infinite number of elements. When z is discrete and its range of values is finite, we can tabulate the elements of A explicitly. For example, if the age in Fig. 3.20 is limited to integers, then A can be written explicitly as

$$A = \{(1,1),(2,1),\ldots,(20,1),(21,0.9),(22,0.8),\ldots,(29,0.1),(30,0),(31,0),\ldots\}$$

Note that, based on the preceding definition, (30, 0) and pairs thereafter are included of A, but their degree of membership in this set is 0. In practice, they typically are not included because interest generally is in elements whose degree of membership is nonzero. Because membership functions determine uniquely the degree of membership in a set, the terms *fuzzy set* and *membership function* are used interchangeably in the literature. This is a frequent source of confusion, so you should keep in mind the routine use of these two

terms to mean the same thing. To help you become comfortable with this terminology, we use both terms interchangeably in this section. When $\mu_A(z)$ can have only two values, say, 0 and 1, the membership function reduces to the familiar characteristic function of ordinary sets. Thus, ordinary sets are a special case of fuzzy sets.

Although fuzzy logic and probability operate over the same [0, 1] interval, there is a significant distinction to be made between the two. Consider the example from Fig. 3.20. A probabilistic statement might read: "There is a 50% chance that a person is young," while a fuzzy statement might read "A person's degree of membership in the set of young people is 0.5." The difference between these two statements is important. In the first statement, a person is considered to be either in the set of young or the set of not young people; we simply have only a 50% chance of knowing to which set the person belongs. The second statement presupposes that a person is young to some degree, with that degree being in this case 0.5. Another interpretation is to say that this is an "average" young person: not really young, but not too near being not young. In other words, fuzzy logic is not probabilistic at all; it just deals with degrees of membership in a set. In this sense, we see that fuzzy logic concepts find application in situations characterized by vagueness and imprecision, rather than by randomness.

The following definitions are basic to the material in the following sections.

Empty set: A fuzzy set is *empty* if and only if its membership function is identically zero in Z.

Equality: Two fuzzy sets A and B are *equal*, written $A = B$, if and only if $\mu_A(z) = \mu_B(z)$ for all $z \in Z$.

The notation "for all $z \in Z$" reads "for all z belonging to Z."

Complement: The *complement* (NOT) of a fuzzy set A, denoted by $\bar{A}$, or NOT(A), is defined as the set whose membership function is

$$\mu_{\bar{A}}(z) = 1 - \mu_A(z)$$

for all $z \in Z$.

Subset: A fuzzy set A is a *subset* of a fuzzy set B if and only if

$$\mu_A(z) \leq \mu_B(z)$$

for all $z \in Z$.

Union: The *union* (OR) of two fuzzy sets A and B, denoted $A \cup B$, or A OR B, is a fuzzy set U with membership function

$$\mu_U(z) = \max\left[\mu_A(z), \mu_B(z)\right]$$

for all $z \in Z$.

Intersection: The *intersection* (AND) of two fuzzy sets A and B, denoted, $A \cap B$ or A AND B, is a fuzzy set I with membership function

$$\mu_I(z) = \min\left[\mu_A(z), \mu_B(z)\right]$$

for all $z \in Z$.

Note that the familiar terms NOT, OR, and AND are used interchangeably with the symbols $\bar{}$, $\cup$, and $\cap$ to denote set complementation, union, and intersection, respectively.

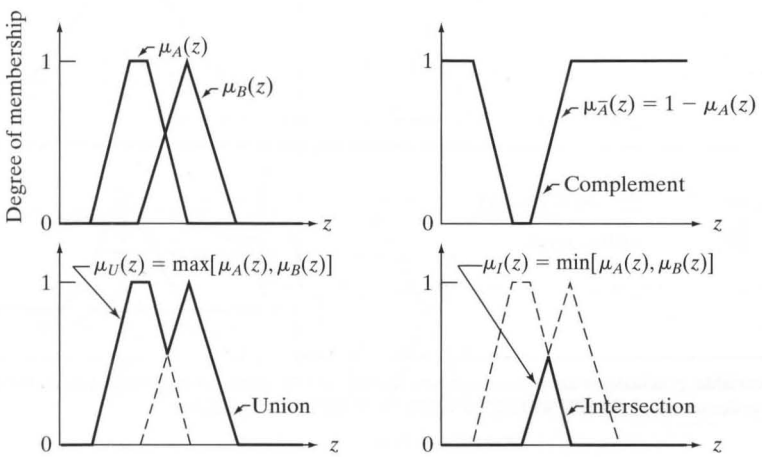
EXAMPLE 3.13: Illustration of fuzzy set definitions.

■ Figure 3.21 illustrates some of the preceding definitions. Figure 3.21(a) shows the membership functions of two sets, A and B, and Fig. 3.21(b) shows the membership function of the complement of A. Figure 3.21(c) shows the membership function of the union of A and B, and Fig. 3.21(d) shows the corresponding result for the intersection of these two sets. The dashed lines in Fig. 3.21are shown for reference only. The results of the fuzzy operations indicated in Figs. 3.21(b)-(d) are the solid lines.

You are likely to encounter examples in the literature in which the area under the curve of the membership function of, say, the intersection of two fuzzy sets, is shaded to indicate the result of the operation. This is a carry over from ordinary set operations and is incorrect. *Only* the points along the membership function itself (solid line) are applicable when dealing with fuzzy sets. This is a good illustration of the comment made earlier that a membership function and its corresponding fuzzy set are one and the same thing. ■

Membership functions

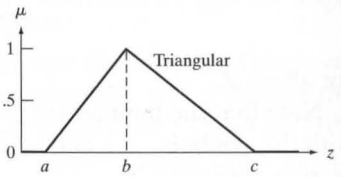
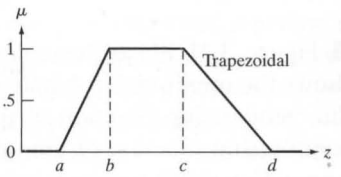
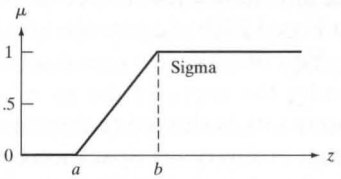
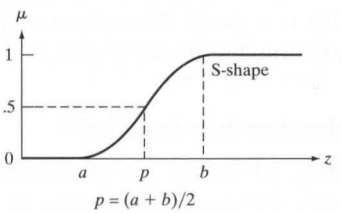
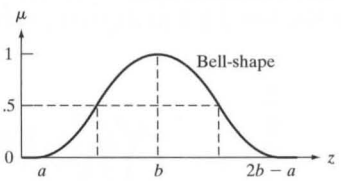
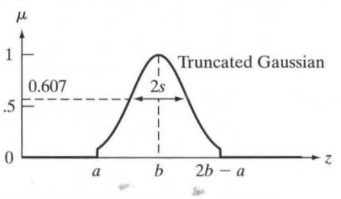
Table 3.6 lists a set of membership functions used commonly for fuzzy set work. The first three functions are piecewise linear, the next two functions are smooth, and the last function is a truncated Gaussian. We develop M-functions in Section 3.6.4 to implement the six membership functions in the table.



a b
c d

FIGURE 3.21
(a) Membership functions of two fuzzy sets, A and B. (b) Membership function of the complement of A. (c) and (d) Membership functions of the union and intersection of A and B.

TABLE 3.6 Some commonly-used membership functions and corresponding plots.

Name	Equation	Plot
Triangular	$\mu(z) = \begin{cases} 0 & z < a \\ (z-a)/(b-a) & a \le z < b \\ 1-(z-b)/(c-b) & b \le z < c \\ 0 & c \le z \end{cases}$	
Trapezoidal	$\mu(z) = \begin{cases} 0 & z < a \\ (z-a)/(b-a) & a \le z < b \\ 1 & b \le z < c \\ 1-(z-b)/(c-b) & c \le z < d \\ 0 & d \le z \end{cases}$	
Sigma	$\mu(z) = \begin{cases} 0 & z < a \\ (z-a)/(b-a) & a \le z < b \\ 1 & b \le z \end{cases}$	
S-shape†	$S(z,a,b) = \begin{cases} 0 & z < a \\ 2\left[\frac{z-a}{b-a}\right]^2 & a \le z < p \\ 1-2\left[\frac{z-b}{b-a}\right]^2 & p \le z < b \\ 1 & b \le z \end{cases}$	
Bell-shape	$\mu(z) = \begin{cases} S(z,a,b) & z < b \\ S(2b-z,a,b) & b \le z \end{cases}$	
Truncated Gaussian	$\mu(z) = \begin{cases} e^{-\frac{(z-b)^2}{s^2}} & \lvert z-b \rvert \le (b-a) \\ 0 & \text{otherwise} \end{cases}$	

†Typically, only the independent variable, z, is used as an argument when writing $\mu(z)$ in order to simplify notation. We made an exception in the S-shape curve in order to use its form in writing the equation of the Bell-shape curve.

3.6.3 Using Fuzzy Sets

In this section we develop the foundation for using fuzzy sets, and then apply the concepts developed here to image processing in Sections 3.6.5 and 3.6.6.

We begin the discussion with an example. Suppose that we want to develop a fuzzy system to monitor the health of an electric motor in a power generating station. For our purposes, the health of the motor is determined by the amount of vibration it exhibits. To simplify the discussion, assume that we can accomplish the monitoring task by using a single sensor that outputs a single number: *average vibration frequency*, denoted by z. We are interested in three ranges of average frequency: *low*, *mid*, and *high*. A motor functioning in the low range is said to be operating *normally*, whereas a motor operating in the mid range is said to be performing *marginally*. A motor whose average vibration is in the high range is said to be operating in the *near-failure* mode.

To simplify notation, we use *frequency* to mean *average vibration frequency* from this point on.

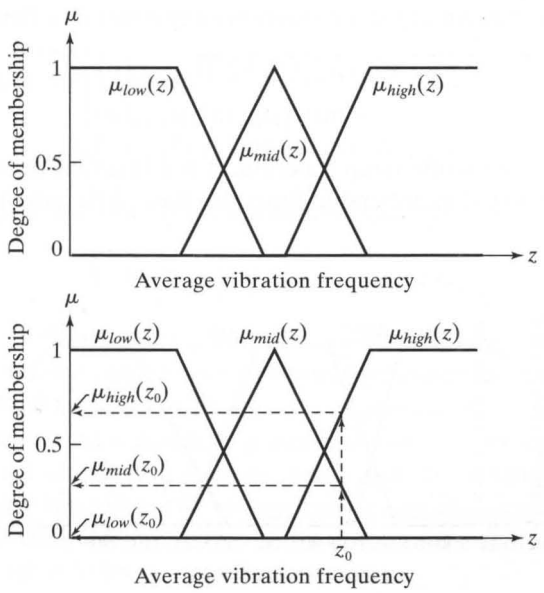
The frequency ranges just discussed may be viewed as fuzzy (in a way similar to age in Fig. 3.20), and we can describe the problem using, for example, the fuzzy membership functions in Fig. 3.22(a). Associating variables with fuzzy membership functions is called *fuzzification*. In the present context, frequency is a *linguistic variable*, and a particular value of frequency, z_0, is called a *linguistic value*. A linguistic value is *fuzzified* by using a membership function to map it to the interval [0, 1]. Figure 3.22(b) shows an example.

Keeping in mind that the frequency ranges are fuzzy, we can express our knowledge about this problem in terms of the following *fuzzy IF-THEN rules*:

R_1: IF the frequency is *low*, THEN motor operation is *normal*.

OR

The part of an if-then rule to the left of THEN is the *antecedent* (or *premise*). The part to the right is called the *consequent* (or *conclusion*.)



a
b

FIGURE 3.22
(a) Membership functions used to fuzzify frequency measurements. (b) Fuzzifying a specific measurement, z_0.

R_2: IF the frequency is *mid*, THEN motor operation is *marginal.*

OR

R_3: IF the frequency is *high*, THEN motor operation is *near failure.*

These rules embody the sum total of our knowledge about the problem; they are simply a formalism for a thought process.

The next step is to find a way to use inputs (frequency measurements) and the knowledge base embodied in the if-then rules to create the outputs of the fuzzy system. This process is called *implication* or *inference*. However, before implication can be applied, the antecedent of each rule has to be processed to yield a *single* value. As we show at the end of this section, multiple parts of an antecedent are linked by ANDs and ORs. Based on the definitions from Section 3.6.2, this means performing min and max operations. To simplify the current explanation, we deal initially with rules whose antecedents contain only one part.

Because we are dealing with fuzzy inputs, the outputs themselves are fuzzy, so membership functions have to be defined for the outputs as well. In this example, the final output in which we are interested is the *percent of operational abnormality* of a motor. Figure 3.23 shows membership functions used to characterize the outputs into three fuzzy classes: *normal*, *marginal*, and *near failure*. Note that the independent variable of the outputs is percent of abnormality (the lower the number the healthier the system), which is different from the independent variable of the inputs.

The membership functions in Figs. 3.22 and 3.23, together with the rule base, contain all the information required to relate inputs and outputs. For example, we note that rule R_1 relates *low* AND *normal*. This is nothing more than the intersection (AND) operation defined earlier. To find the result of the AND operation between these two functions, recall from Section 3.6.2 that AND is defined as the minimum of the two membership functions; that is,

$$\begin{aligned}\mu_1(z, v) &= \mu_{low}(z) \text{ AND } \mu_{norm}(v) \\ &= \min\{\mu_{low}(z), \mu_{norm}(v)\}\end{aligned}$$

This result also is a membership function. It is a function of two variables because the two ANDed membership functions have different independent variables.

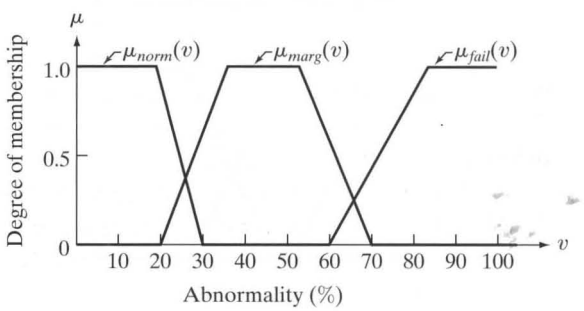


FIGURE 3.23 Membership functions used for characterizing the fuzzy conditions *normal*, *marginal*, and *near failure*.

The preceding equation is a general result. We are interested in outputs due to *specific* inputs. Let z_0 denote a specific value of frequency. The degree of membership of this input in terms of the *low* membership function is $\mu_{low}(z_0)$. We find the output corresponding to rule R_1 and input z_0 by ANDing $\mu_{low}(z_0)$ and the general result $\mu_1(z, v)$ evaluated at z_0:

$$\begin{aligned} Q_1(v) &= \min\{\mu_{low}(z_0), \mu_1(z_0, v)\} \\ &= \min\{\mu_{low}(z_0), \min\{\mu_{low}(z_0), \mu_{norm}(v)\}\} \\ &= \min\{\mu_{low}(z_0), \mu_{norm}(v)\} \end{aligned}$$

where the last step follows by inspection from the fact that $\mu_{low}(z_0)$ is a constant [see Fig. 3.22(b)]. Here, $Q_1(v)$ denotes the *fuzzy* output due to rule R_1 and a specific input. The only variable in Q_1 is the output variable v, as expected.

Following the same line of reasoning, we arrive at the following outputs due to the other two rules and the same specific input:

$$Q_2(v) = \min\{\mu_{mid}(z_0), \mu_{marg}(v)\}$$

and

$$Q_3(v) = \min\{\mu_{high}(z_0), \mu_{fail}(v)\}$$

Each of the preceding three equations is the output associated with a particular rule and a specific input. Each of these responses is a fuzzy set, despite the fact that the input is a fixed value. The procedure just described is the implication process mentioned a few paragraphs back, which yields the output due to inputs and the knowledge embodied in the if-then rules.

To obtain the overall response of the fuzzy system we *aggregate* the three individual responses. In the rule base given at the beginning of this section the three rules are associated by the OR (*union*) operation. Thus, the complete (aggregated) fuzzy output is given by

$$Q(v) = Q_1(v) \text{ OR } Q_2(v) \text{ OR } Q_3(v)$$

Because OR is defined as a max operation, we can write this result as

$$Q(v) = \max_r \left\{ \min_{s,t} \{\mu_s(z_0), \mu_t(v)\} \right\}$$

for $r = \{1, 2, 3\}$, $s = \{low, mid, high\}$, and $t = \{norm, marg, fail\}$. It is implied that values of s and t are paired in valid combinations (i.e., *low* and *norm*, *mid* and *marg*, and *high* and *fail*). Although it was developed in the context of an example, this expression is perfectly general; to extend it to n rules we simply let $r = \{1, 2, \ldots, n\}$; similarly, we can expand s and t to include any finite number of membership functions. The two preceding equations say the same thing: The response, Q, of our fuzzy system is the union of the individual fuzzy sets resulting from each rule by the implication process.

Figure 3.24 summarizes the results to this point. Figure 3.24(a) contains the elements needed to compute the individual outputs Q_1, Q_2, and Q_3: (1) $\mu_s(z_0)$ for $s = \{low, mid, high\}$ (these are constants—see Fig. 3.22); and (2) functions $\mu_t(v)$ for $t = \{norm, marg, fail\}$. The Q_i are obtained by computing the minimum between corresponding pairs of these quantities. Figure 3.24(b) shows the result. Note that the net effect of computing the minimum between each $\mu_t(v)$ and its corresponding constant $\mu_s(z_0)$ is nothing more than *clipping* $\mu_t(v)$ at the value of $\mu_s(z_0)$. Finally, we obtain a single (aggregated) output, $Q(v)$, by computing the maximum value between all three $Q_i(v)$ at each value of v. Figure 3.24(c) shows the result.

We have successfully obtained the complete output corresponding to a specific input, but we still are dealing with a fuzzy set, $Q(v)$. The last step is to obtain a crisp output, v_0, from fuzzy set Q using a process appropriately called *defuzzification*.

a
b
c

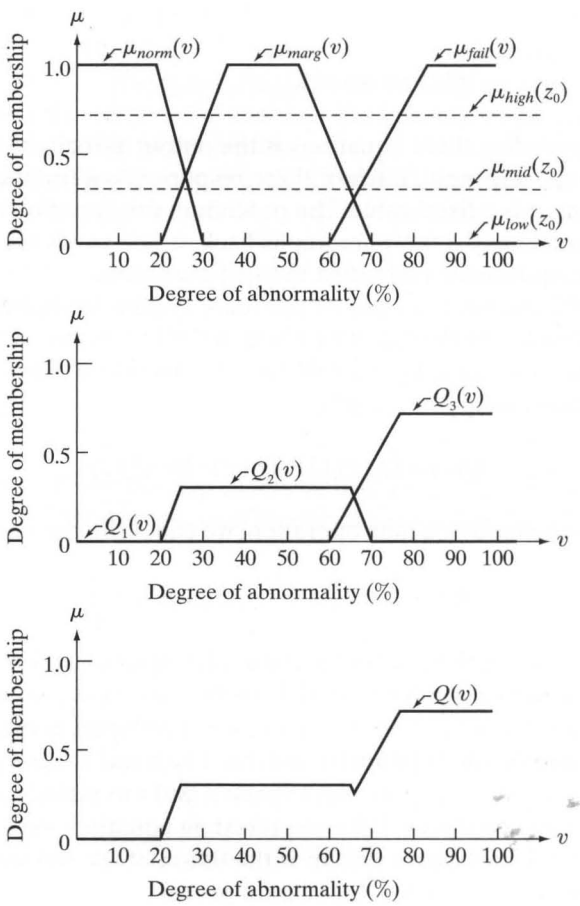


FIGURE 3.24
Values of μ_{low}, μ_{mid}, and μ_{high} evaluated at z_0 (all three are constant values). (b) Individual outputs. (c) Aggregated output.

There are a number of ways to defuzzify Q to obtain a crisp output v_0. A common approach is to compute the center of gravity of $Q(v)$:

$$v_0 = \frac{\int vQ(v)\,dv}{\int Q(v)\,dv}$$

In theory, Q and v can be continuous. When performing defuzzification, the approach typically is to define a set of discrete values for v and approximate the integrals by summations. Function `defuzzify` in Section 3.6.4 does this.

where the integrals are taken over the range of values of the independent variable, v. In Example 3.14 (Section 3.6.4) we illustrate how to approximate this function by summations, using Q from Fig. 3.24(c) and a specific frequency value $z_0 = 0.7$ [see Fig. 3.22(b)]. The result is $v_0 = 0.76$, meaning that, for that frequency value, the motor is operating with a 76% degree of abnormality.

Thus far, we have considered if-then rules whose antecedents have only one part, such as "IF the frequency is *low*." Rules with antecedents that have more than one part must be combined to yield a *single* number that represents the entire antecedent for that rule. For example, suppose that we have the rule: IF the frequency is *low* AND the temperature is *moderate* THEN motor operation is *normal*. A membership function would have to be defined for the linguistic variable *moderate*. Then, to obtain a single number for this rule that takes into account both parts of the antecedent, we first evaluate a given value of frequency using the *low* membership function and a given value of *temperature* using the *moderate* membership function. Because the two parts are linked by AND, we use the minimum of the two resulting values. This value is then used in the implication process to "clip" the *normal* output membership function, which is the function associated with this rule. The rest of the procedure is as before, as the following summary illustrates.

Figure 3.25 shows the motor example using two inputs, *frequency* and *temperature*. We can use this figure and the preceding material to summarize the principal steps followed in the application of rule-based fuzzy logic:

1. *Fuzzify the inputs:* For each scalar input, find the corresponding fuzzy values by mapping that input to the interval [0, 1] using the applicable membership functions in each rule, as the first two columns of Fig. 3.25 show.
2. *Perform any required fuzzy logical operations:* The outputs of all parts of an antecedent must be combined to yield a *single* value using the max or min operation, depending on whether the parts are connected by ORs or by ANDs. In Fig. 3.25 all the parts of the antecedents are connected by ANDs, so the min operation is used throughout. The number of parts of an antecedent and the type of logic operator used to connect them can be different from rule to rule.
3. *Apply an implication method:* The single output of the antecedent of each rule is used to provide the output corresponding to that rule. As explained earlier, the method of implication we are using is based on ANDs, which are min operations. This clips the corresponding output membership function at the value provided by the antecedent, as the third and fourth columns in Fig. 3.25 show.

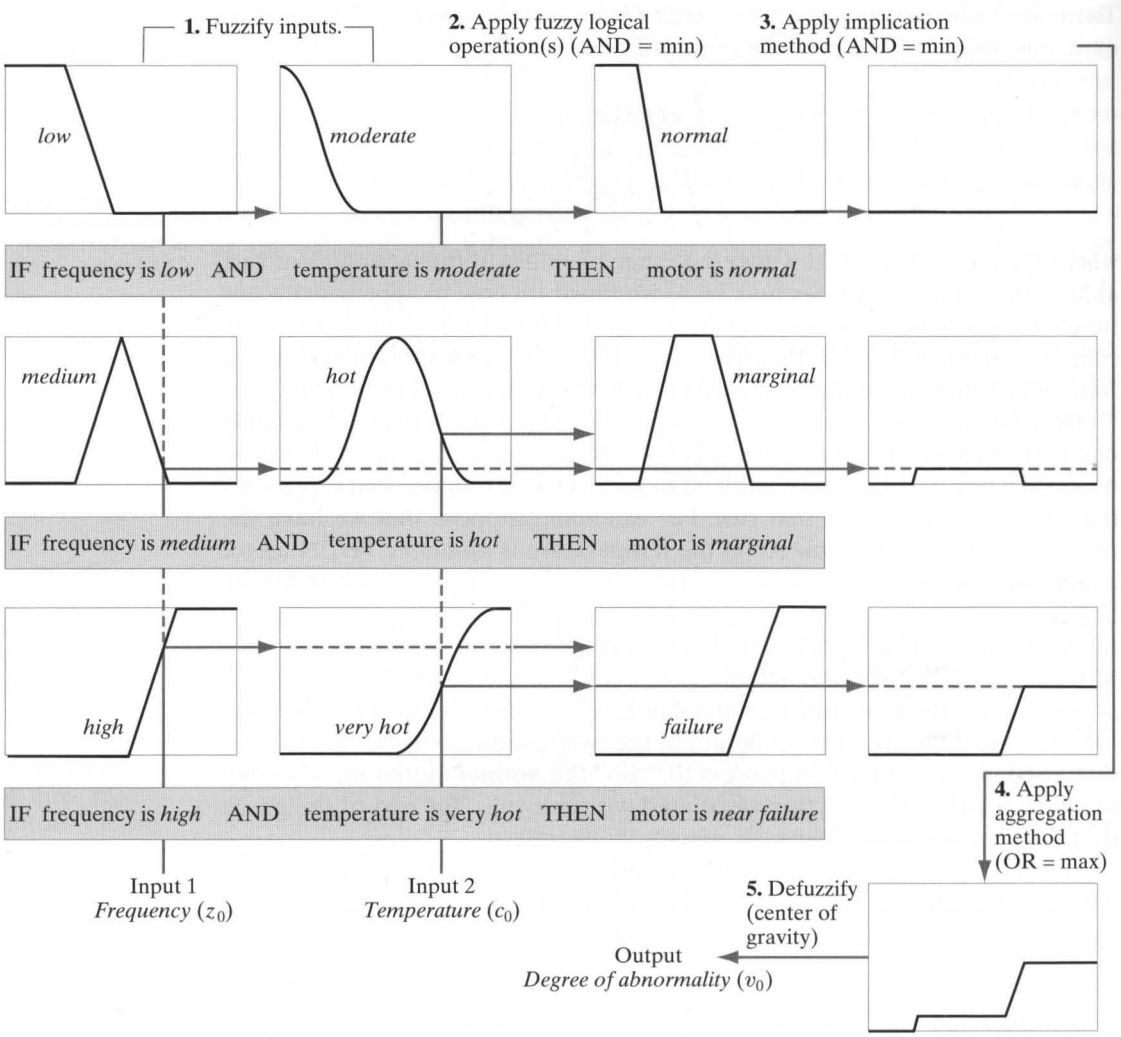


FIGURE 3.25 Example illustrating the five basic steps used typically to implement a fuzzy rule-based system: (1) fuzzification, (2) logical operations, (3) implication, (4) aggregation, and (5) defuzzification.

4. *Apply an aggregation method to the fuzzy sets from step 3:* As the last column in Fig. 3.25 shows, the output of each rule is a fuzzy set. These must be combined to yield a single output fuzzy set. The approach used here is to form the union (OR) of the individual outputs, so the max operation is employed.
5. *Defuzzify the final output fuzzy set:* In this final step we obtain a crisp, scalar output. This is achieved by computing the center of gravity of the aggregated set from step 4.

When the number of variables is large, it is advantageous to use the shorthand notation (*variable*, *fuzzy set*) to pair a variable with its corresponding membership function. For example, the rule "IF the frequency value is *low*, THEN motor operation is *normal*" would be written as: "IF (z, *low*) THEN (v, *normal*)" where, as before variables z and v represent average frequency and percent abnormality, respectively, while *low* and *normal* are defined by the two membership functions $\mu_{low}(z)$ and $\mu_{norm}(v)$, respectively.

In general, when working with M if-then rules, N input variables, $z_1, z_2, \ldots, z_N$, and one output variable, v, the type of fuzzy rule formulation used most frequently in image processing has the form

$$\text{IF } (z_1, A_{11}) \text{ AND } (z_2, A_{12}) \text{ AND } \ldots \text{ AND } (z_N, A_{1N}) \text{ THEN } (v, B_1)$$
$$\text{IF } (z_1, A_{21}) \text{ AND } (z_2, A_{22}) \text{ AND } \ldots \text{ AND } (z_N, A_{2N}) \text{ THEN } (v, B_2)$$
$$\ldots\ldots$$
$$\text{IF } (z_1, A_{M1}) \text{ AND } (z_2, A_{M2}) \text{ AND } \ldots \text{ AND } (z_N, A_{MN}) \text{ THEN } (v, B_M)$$
$$\text{ELSE } (v, B_E)$$

where A_{ij} is the fuzzy set associated with the ith rule and the jth input variable, B_i is the fuzzy set associated with the output of the ith rule, and we have assumed that the components of the rule antecedents are linked by ANDs. Note that we introduced an ELSE rule, with associated fuzzy set B_E. This rule is executed when none of the preceding rules is satisfied completely; its output is explained below.

As indicated earlier, all the elements of the antecedent of each rule are evaluated to yield a single scalar value. In Fig. 3.25 we used the min operation because the rules are based on ANDs. The preceding general formulation also uses ANDs, so we use the min operator again. Evaluating the antecedents of the ith rule produces a scalar output, λ_i, given by

$$\lambda_i = \min\left\{\mu_{A_{ij}}(z_j);\ j = 1, 2, \ldots, N\right\}$$

for $i = 1, 2, \ldots, M$, where $\mu_{A_{ij}}(z_j)$ is the membership function of fuzzy set A_{ij} evaluated at the value of the jth input. Often, λ_i is called the *strength level* (or *firing level*) of the ith rule. We know from our earlier discussion that λ_i is simply the value used to clip the output function of the ith rule.

The ELSE rule is executed when the conditions of the THEN rules are weakly satisfied (we give in Section 3.6.6 a detailed example of how ELSE rules are used). The ELSE response should be strong when all the others are weak. In a sense, you can view an ELSE rule as performing a NOT operation on the results of the other rules. We know from Section 3.6.2 that

$$\mu_{\text{NOT}(A)}(z) = \mu_{\bar{A}}(z) = 1 - \mu_A(z)$$

Then, using this idea in combining (ANDing) all the levels of the THEN rules, gives the following strength level for the ELSE rule:

$$\lambda_E = \min\left\{1 - \lambda_i;\ i = 1, 2, \ldots, M\right\}$$

We see that if all the THEN rules fire at "full strength" (all their responses are 1) then the response of the ELSE rule is 0, as expected. As the responses of the THEN rules weaken, the strength of the ELSE rule increases. This is the fuzzy counterpart of the familiar if-then-else rule used in software programming.

When dealing with ORs in the antecedents, we simply replace the ANDs in the general formulation given earlier by ORs and the min in the equation for λ_i by a max; the expression for λ_E does not change. Although we could formulate more complex antecedents and consequents than the ones discussed here, the formulations we have developed using only ANDs or ORs are quite general and are used in a broad spectrum of image processing applications. Implementation of fuzzy methods tends to be computationally intensive, so fuzzy formulations should be kept as simple as possible.

3.6.4 A Set of Custom Fuzzy M-functions

In this section we develop a set of M-functions that implement all the membership functions in Table 3.6 and generalize the model summarized in Fig. 3.25. As such, these functions can be used as the basis for the design of a broad class of rule-based fuzzy systems. Later in this section, we use these functions to compute the output of the motor monitoring system discussed in the previous section. Then, in Sections 3.6.5 and 3.6.6, we illustrate how to expand the functionality of the functions by applying them to fuzzy intensity transformations and spatial filtering.

MATLAB nested functions

We use nested functions extensively in the following sections, so we digress briefly to study this important concept. Nested functions are a relatively new programming feature introduced in MATLAB 7. In the context of this section, our interest in nested functions is in the formulation of function-generating functions which, as you will see shortly, are well suited for the types of functions used in fuzzy processing.

A *nested function* is a function defined within the body of another function. When an M-file contains nested functions, all functions in the file must be terminated with the `end` keyword. For example, a function containing one nested function has the following general syntax:

```
function [outputs1] = outer_function(arguments1)
statements

   function [outputs2] = inner_function(arguments2)
      statements
   end

statements
end
```

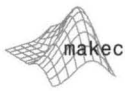
A variable used or defined in a nested function resides in the workspace of the outermost function that both contains the nested function and accesses that variable. For example

```
function y = tax(income)
adjusted_income = income - 6000;
y = compute_tax
   function y = compute_tax
      y = 0.28 * adjusted_income;
   end
end
```

The variable `adjusted_income` appears in the nested function `compute_tax` and it also appears in the enclosing function `tax`. Therefore, both instances of `adjusted_income` refer to the *same* variable.

When you form a handle to a nested function, the workspace of variables of that function are incorporated into the handle, and the workspace variables continue to exist as long as the function handle exists. The implication is that functions handles can be created that can access and modify the contents of their own workspace. This feature makes possible the creation of *function-generating functions* (also called *function factories*). For example, MATLAB ships with a demo function that makes a function capable of determining how many times it has been called:

See Section 2.10.4 regarding function handles.




```
function countfcn = makecounter(initvalue)
%MAKECOUNTER Used by NESTEDDEMO.
%   This function returns a handle to a customized nested function
%   'getCounter'.
%   initvalue specifies the initial value of the counter whose handle
%   is returned.
%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 1.1.6.2 $ $Date: 2004/03/02 21:46:55 $

currentCount = initvalue; % Initial value.
countfcn = @getCounter; % Return handle to getCounter.

   function count = getCounter
      % This function increments the variable 'currentCount', when it
      % is called (using its function handle).
      currentCount = currentCount + 1;
      count = currentCount;
   end

end
```

The output from `makecounter` is a function handle to the nested function `getCounter`. Whenever it is called, this function handle can access the variable workspace of `getCounter`, including the variable `currentCount`. When called, `getCounter` increments this variable and then returns its value. For example,

```
>> f = makecounter(0); % Set initial value.
>> f()
```

Recall from Section 2.10.4 that you use ( ) to call a function handle with no input arguments.

```
ans =
     1

>> f()
ans =
     2

>> f()
ans =
     3
```

As is true of any language that supports recursive function calls, separate calls to a function in MATLAB result in separate instances of that function's variables. This means that a function-generating function, when called multiple times, makes functions that have independent states. For example, you can make more than one counter function, each of which maintains a count that is independent of the others:

```
f1 = makecounter(0);
f2 = makecounter(20);
f1()
ans =
     1

f2()
ans =
     21
```

Several of the fuzzy functions we develop later in this section accept one set of functions as inputs and produce another set of functions as outputs. The following code introduces this concept:

```
function h = compose(f, g)
h = @composeFcn;
   function y = composeFcn(x)
      y = f(g(x));
   end
end
```

where `f` and `g` are function handles. Function `compose` takes these two handles as inputs and returns a *new* function handle, `h`, that is their composition, defined in this case as `h(x) = f(g(x))`. For example, consider the following:

```
>> g = @(x) 1./x;
```

```
>> f = @sin;
```

Letting

```
>> h = compose(f, g);
```

results in the function `h(x) = sin(1./x)`. Working with the new function handle `h` is the same as working with `sin(1./x)`. For instance, to plot this function in the interval `[-1, 1]`, we write

```
>> fplot(h, [-1 1], 20) % See Section 3.3.1 regarding fplot.
```

We use the ideas just introduced later in this section, starting with function `lambdafcns`.

Membership functions

The following M-functions are self-explanatory. They are direct implementation of the equations of the membership functions in Table 3.6. In fact, the plots in that table were generated using these functions. Observe that all functions are vectorized, in the sense that the independent variable, `z`, can be a vector of any length.

triangmf

```
function mu = triangmf(z, a, b, c)
%TRIANGMF Triangular membership function.
%   MU = TRIANGMF(Z, A, B, C) computes a fuzzy membership function
%   with a triangular shape. Z is the input variable and can be a
%   vector of any length. A, B, and C are scalar parameters, such
%   that B >= A and C >= B, that define the triangular shape.
%
%       MU = 0,                         Z < A
%       MU = (Z - A) ./ (B - A),        A <= Z < B
%       MU = 1 - (Z - B) ./ (C - B),    B <= Z < C
%       MU = 0,                         C <= Z

mu = zeros(size(z));

low_side  = (a <= z) & (z < b);
high_side = (b <= z) & (z < c);

mu(low_side)  = (z(low_side) - a) ./ (b - a);
mu(high_side) = 1 - (z(high_side) - b) ./ (c - b);
```

trapezmf

```
function mu = trapezmf(z, a, b, c, d)
%TRAPEZMF Trapezoidal membership function.
%   MU = TRAPEZMF(Z, A, B, C) computes a fuzzy membership function
%   with a trapezoidal shape. Z is the input variable and can be a
%   vector of any length. A, B, C, and D are scalar parameters that
%   define the trapezoidal shape. The parameters must be ordered so
%   that A <= B, B <= C, and C <= D.
```

```
%
%       MU = 0,                          Z < A
%       MU = (Z - A) ./ (B - A),         A <= Z < B
%       MU = 1,                          B <= Z < C
%       MU = 1 - (Z - C) ./ (D - C),     C <= Z < D
%       MU = 0,                          D <= Z

mu = zeros(size(z));

up_ramp_region = (a <= z) & (z < b);
top_region = (b <= z) & (z < c);
down_ramp_region = (c <= z) & (z < d);

mu(up_ramp_region) = 1 - (b - z(up_ramp_region)) ./ (b - a);
mu(top_region) = 1;
mu(down_ramp_region) = 1 - (z(down_ramp_region) - c) ./ (d - c);
```

sigmamf

```
function mu = sigmamf(z, a, b)
%SIGMAMF Sigma membership function.
%   MU = SIGMAMF(Z, A, B) computes the sigma fuzzy membership
%   function. Z is the input variable and can be a vector of
%   any length. A and B are scalar shape parameters, ordered
%   such that A <= B.
%
%       MU = 0,                          Z < A
%       MU = (Z - A) ./ (B - A),         A <= Z < B
%       MU = 1,                          B <= Z
```

Note how the sigma function is generated as a special case of the trapezoidal function.

```
mu = trapezmf(z, a, b, Inf, Inf);
```

smf

```
function mu = smf(z, a, b)
%SMF S-shaped membership function.
%   MU = SMF(Z, A, B) computes the S-shaped fuzzy membership
%   function. Z is the input variable and can be a vector of any
%   length. A and B are scalar shape parameters, ordered such that
%   A <= B.
%
%       MU = 0,                              Z < A
%       MU = 2*((Z - A) ./ (B - A)).^2,      A <= Z < P
%       MU = 1 - 2*((Z - B) ./ (B - A)).^2,  P <= Z < B
%       MU = 1,                              B <= Z
%
%   where P = (A + B)/2.

mu = zeros(size(z));

p = (a + b)/2;
low_range = (a <= z) & (z < p);
mu(low_range) = 2 * ( (z(low_range) - a) ./ (b - a) ).^2;
```

```
mid_range = (p <= z) & (z < b);
mu(mid_range) = 1 - 2 * ( (z(mid_range) - b) ./ (b - a) ).^2;

high_range = (b <= z);
mu(high_range) = 1;
```

bellmf

```
function mu = bellmf(z, a, b)
%BELLMF Bell-shaped membership function.
%   MU = BELLMF(Z, A, B) computes the bell-shaped fuzzy membership
%   function. Z is the input variable and can be a vector of any
%   length. A and B are scalar shape parameters, ordered such that
%   A <= B.
%
%       MU = SMF(Z, A, B),          Z < B
%       MU = SMF(2*B - Z, A, B),    B <= Z

mu = zeros(size(z));

left_side = z < b;
mu(left_side) = smf(z(left_side), a, b);

right_side = z >= b;
mu(right_side) = smf(2*b - z(right_side), a, b);
```

Note how this function is generated as two halves of the S-shape function.

truncgaussmf

```
function mu = truncgaussmf(z, a, b, s)
%TRUNCGAUSSMF Truncated Gaussian membership function.
%   MU = TRUNCGAUSSMF(Z, A, B, S) computes a truncated Gaussian
%   fuzzy membership function. Z is the input variable and can be a
%   vector of any length. A, B, and S are scalar shape parameters. A
%   and B have to be ordered such that A <= B.
%
%       MU = exp(-(Z - B).^2 / s^2),  abs(Z - B) <= (B - A)
%       MU = 0,                       otherwise

mu = zeros(size(z));

c = a + 2*(b - a);
range = (a <= z) & (z <= c);
mu(range) = exp(-(z(range) - b).^2 / s^2);
```

The following utility functions are used in situations in which it is necessary for a rule to have no effect on the output. We give an example of this in Section 3.6.6.

zeromf

```
function mu = zeromf(z)
%ZEROMF Constant membership function (zero).
%   ZEROMF(Z) returns an an array of zeros with the same size as Z.
%
```

```
%   When using the @max operator to combine rule antecedents,
%   associating this membership function with a particular input
%   means that input has no effect.

mu = zeros(size(z));
```

onemf

```
function mu = onemf(z)
%ONEMF Constant membership function (one).
%   ONEMF(Z) returns an an array of ones with the same size as Z.
%
%   When using the @min operator to combine rule antecedents,
%   associating this membership function with a particular input
%   means that input has no effect.

mu = ones(size(z));
```

Function for computing rule strengths

Once the input and output membership functions have been defined using any of the preceding M-functions, the next step is to evaluate the rules for any given input. That is, we compute the rule strengths (the lambda functions defined in the previous section), which is the implementation of the first two steps in the procedure outlined in Section 3.6.3. The following function, `lambdafcns`, performs this task. Observe that using nested functions allows `lambdafcns` to output a set of lambda *functions* instead of numerical outputs. We could, for instance, plot the outputs of the function. An analogy from mathematics is to write a set of equations in terms of variables instead of specific values. This capability would be difficult to implement without nested functions.

lambdafcns

```
function L = lambdafcns(inmf, op)
%LAMBDAFCNS Lambda functions for a set of fuzzy rules.
%   L = LAMBDAFCNS(INMF, OP) creates a set of lambda functions
%   (rule strength functions) corresponding to a set of fuzzy rules.
%   L is a cell array of function handles. INMF is an M-by-N matrix
%   of input membership function handles. M is the number of rules,
%   and N is the number of fuzzy system inputs. INMF(i, j) is the
%   input membership function applied by the i-th rule to the j-th
%   input. For example, in the case of Fig. 3.25, INMF would be of
%   size 3-by-2 (three rules and two inputs).
%
%   OP is a function handle used to combine the antecedents for each
%   rule. OP can be either @min or @max. If omitted, the default
%   value for OP is @min.
%
%   The output lambda functions are called later with N inputs,
%   Z1, Z2, ..., ZN, to determine rule strength:
%
%       lambda_i = L{i}(Z1, Z2, ..., ZN)

if nargin < 2
```

See Section 2.10.7 regarding cell arrays.

```
    % Set default operator for combining rule antecedents.
    op = @min;
end

num_rules = size(inmf, 1);
L = cell(1, num_rules);

for i = 1:num_rules
    % Each output lambda function calls the ruleStrength() function
    % with i (to identify which row of the rules matrix should be
    % used), followed by all the Z input arguments (which are passed
    % along via varargin).
    L{i} = @(varargin) ruleStrength(i, varargin{:});
end

    %-------------------------------------------------------------%
    function lambda = ruleStrength(i, varargin)
        % lambda = ruleStrength(i, Z1, Z2, Z3, ...)
        Z = varargin;
        % Initialize lambda as the output of the first membership
        % function of the k-th rule.
        memberfcn = inmf{i, 1};
        lambda = memberfcn(Z{1});
        for j = 2:numel(varargin)
            memberfcn = inmf{i, j};
            lambda = op(lambda, memberfcn(Z{j}));
        end
    end

end
```

Function for performing implications

Implication is the next step in the procedure outlined in Section 3.6.3. Implication requires the specific the response of each rule and a set of corresponding output membership functions. The output of function `lambdafcns` provides rule strengths in "general" terms. Here we need to provide specific inputs to be able to carry out implication. The following function uses nested functions to produce the required implication functions. As before, the use of nested functions allows the generation of the implication functions themselves (see the fourth column in Fig. 3.25).

```
function Q = implfcns(L, outmf, varargin)                    implfcns
%IMPLFCNS Implication functions for a fuzzy system.
%   Q = IMPLFCNS(L, OUTMF, Z1, Z2, ..., ZN) creates a set of
%   implication functions from a set of lambda functions L, a set of
%   output member functions OUTMF, and a set of fuzzy system inputs
%   Z1, Z2, ..., ZN. L is a cell array of rule-strength function
%   handles as returned by LAMBDAFCNS. OUTMF is a cell array of
```

```
%   output membership functions. The number of elements of OUTMF can
%   either be numel(L) or numel(L)+1. If numel(OUTMF) is numel(L)+1,
%   then the "extra" membership function is applied to an
%   automatically computed "else rule." (See Section 3.6.3.). The
%   inputs Z1, Z2, etc., can all be scalars, or they can all be
%   vectors of the same size (i.e., these vectors would contain
%   multiple values for each of the inputs).
%
%   Q is a 1-by-numel(OUTMF) cell array of implication function
%   handles.
%
%   Call the i-th implication function on an input V using the
%   syntax:
%
%       q_i = Q{i}(V)

Z = varargin;

% Initialize output cell array.
num_rules = numel(L);
Q = cell(1, numel(outmf));
lambdas = zeros(1, num_rules);

for i = 1:num_rules
   lambdas(i) = L{i}(Z{:});
end

for i = 1:num_rules
   % Each output implication function calls implication() with i (to
   % identify which lambda value should be used), followed by V.
   Q{i} = @(v) implication(i, v);
end

if numel(outmf) == (num_rules + 1)
   Q{num_rules + 1} = @elseRule;
end

   %-------------------------------------------------------------------%
   function q = implication(i, v)
      q = min(lambdas(i), outmf{i}(v));
   end

   %-------------------------------------------------------------------%
   function q = elseRule(v)
      lambda_e = min(1 - lambdas);
      q = min(lambda_e, outmf{end}(v));
   end

end
```

Function for performing aggregation

The next step in our procedure is to aggregate the functions resulting from implication. Again using nested functions allows us to write code that outputs the aggregated function itself (see the function at the bottom of the fourth column in Fig. 3.25).

aggfcn

```
function Qa = aggfcn(Q)
%AGGFCN Aggregation function for a fuzzy system.
%   QA = AGGFCN(Q) creates an aggregation function, QA, from a
%   set of implication functions, Q. Q is a cell array of function
%   handles as returned by IMPLFCNS. QA is a function handle that
%   can be called with a single input V using the syntax:
%
%       q = QA(V)

Qa = @aggregate;

    function q = aggregate(v)
        q = Q{1}(v);
        for i = 2:numel(Q)
            q = max(q, Q{i}(v));
        end
    end
end
```

Function for performing defuzzification

The output of `aggfcn` is a fuzzy function. To get the final, crisp output, we perform defuzzification, as explained in Section 3.6.3. The following function does this. Note that the output in this case is a *numerical* value, as opposed to the outputs of `lambdafcns`, `implfcns`, and `aggfcn`, which are functions. Note also that no nested functions were needed here.

defuzzify

This function shows one approach to approximating the integral form of the center of gravity introduced in Section 3.6.3 to obtain a defuzzified scalar value.

```
function out = defuzzify(Qa, vrange)
%DEFUZZIFY Output of fuzzy system.
%   OUT = DEFUZZIFY(QA, VRANGE) transforms the aggregation function
%   QA into a fuzzy result using the center-of-gravity method. QA is
%   a function handle as returned by AGGFCN. VRANGE is a two-element
%   vector specifying the range of input values for QA. OUT is the
%   scalar result.

v1 = vrange(1);
v2 = vrange(2);

v = linspace(v1, v2, 100);
Qv = Qa(v);
out = sum(v .* Qv) / sum(Qv);
if isnan(out)
   % If Qv is zero everywhere, out will be NaN. Arbitrarily choose
```

```
    % output to be the midpoint of vrange.
    out = mean(vrange);
end
```

Putting it all together

The following function combines the preceding fuzzy functions into a single M-file that accepts a set of input and output membership functions and yields a single fuzzy system function that can be evaluated for any set of inputs. In other words, the following function generalizes and integrates the entire process summarized in Fig. 3.25. As you will see in Example 3.14, and in Sections 3.6.5 and 3.6.6, the effort required to design of a fuzzy system is reduced considerably by using this function.

fuzzysysfcn

```
function F = fuzzysysfcn(inmf, outmf, vrange, op)
%FUZZYSYSFCN Fuzzy system function.
%   F = FUZZYSYSFCN(INMF, OUTMF, VRANGE, OP) creates a fuzzy system
%   function, F, corresponding to a set of rules and output
%   membership functions. INMF is an M-by-N matrix of input
%   membership function handles. M is the number of rules, and N is
%   the number of fuzzy system inputs. OUTMF is a cell array
%   containing output membership functions. numel(OUTMF) can be
%   either M or M + 1. If it is M + 1, then the "extra" output
%   membership function is used for an automatically computed "else
%   rule." VRANGE is a two-element vector specifying the valid range
%   of input values for the output membership functions. OP is a
%   function handle specifying how to combine the antecedents for
%   each rule. OP can be either @min or @max. If OP is omitted, then
%   @min is used.
%
%   The output, F, is a function handle that computes the fuzzy
%   system's output, given a set of inputs, using the syntax:
%
%       out = F(Z1, Z2, Z3, ..., ZN)

if nargin < 4
   op = @min;
end

% The lambda functions are independent of the inputs Z1, Z2, ...,
% ZN, so they can be computed in advance.
L = lambdafcns(inmf, op);

F = @fuzzyOutput;

   %-------------------------------------------------------------%
   function out = fuzzyOutput(varargin)
      Z = varargin;
      % The implication functions and aggregation functions have to
```

```
        % be computed separately for each input value.  Therefore we
        % have to loop over each input value to determine the
        % corresponding output value. Zk is a cell array that will be
        % used to pass scalar values for each input (Z1, Z2, ..., ZN)
        % to IMPLFCNS.
        Zk = cell(1, numel(Z));
        % Initialize the array of output values to be the same size as
        % the first input, Z{1}.
        out = zeros(size(Z{1}));
        for k = 1:numel(Z{1})
           for p = 1:numel(Zk)
              Zk{p} = Z{p}(k);
           end
           Q = implfcns(L, outmf, Zk{:});
           Qa = aggfcn(Q);
           out(k) = defuzzify(Qa, vrange);
        end
     end

end
```

Improving performance

The fuzzy system function created by `fuzzysysfcn` gives the exact output for any set of inputs. Although it is useful for exploration and plotting purposes, it is too slow for large inputs, as is typical in image processing. Function `approxfcn` creates an approximation to the fuzzy system function. The approximation uses a lookup table and runs much faster.

When a function takes more than a few seconds to execute, it is a good practice to provide a visual cue to the user, indicating percent completion. MATLAB's function `waitbar` is used for that purpose. The syntax

```
h = waitbar(c, 'message')
```

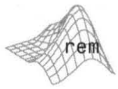


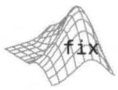
displays a waitbar of fractional length `c`, where `c` is between 0 and 1. A typical application (which is the one we use here) is to place a waitbar inside a `for` loop that performs a lengthy computation. The following code fragment illustrates how this is done:

```
h = waitbar(0, 'Working. Please wait . . . '); % Initialize.
for I = 1:L
   % Computations go here %
   waitbar(I/L) % Update the progress bar.
end
close(h)
```

The computational overhead inherent in updating the bar during each pass through a loop can be reduced by updating the bar periodically. The following modifies the preceding code fragment to update the bar at 2% intervals:

```
h = waitbar(0, 'Working. Please wait . . . '); % Initialize.
waitbar_update_interval = ceil(0.02 * L)
for I = 1:L
   % Computations go here %
   % Check progress.
   if rem(I, waitbar_update_interval) == 0)
      waitbar(I/L)
   end
end
close(h)
```





where `rem(X, Y) = X - fix(X./Y)*Y` and `fix(X./Y)` gives the integer part of the division.

approxfcn

```
function G = approxfcn(F, range)
%APPROXFCN Approximation function.
%   G = APPROXFCN(F, RANGE) returns a function handle, G, that
%   approximates the function handle F by using a lookup table.
%   RANGE is an M-by-2 matrix specifying the input range for each of
%   the M inputs to F.

num_inputs = size(range, 1);
max_table_elements = 10000;
max_table_dim = 100;
table_dim = min(floor(max_table_elements^(1/num_inputs)), ...
   max_table_dim);

% Compute the input grid values.
inputs = cell(1, num_inputs);
grid = cell(1, num_inputs);
for k = 1:num_inputs
   grid{k} = linspace(range(k, 1), range(k, 2), table_dim);
end

if num_inputs > 1
   [inputs{:}] = ndgrid(grid{:});
else
   inputs = grid;
end

% Initialize the lookup table.
table = zeros(size(inputs{1}));

% Initialize waitbar.
bar = waitbar(0,'Working...');

% Initialize cell array used to pass inputs to F.
Zk = cell(1, num_inputs);
L = numel(inputs{1});
% Update the progress bar at 2% intervals.
```

```
for p = 1:L
   for k = 1:num_inputs
      Zk{k} = inputs{k}(p);
   end
   table(p) = F(Zk{:});
   if (rem(p, waitbar_update_interval) == 0)
    % Update the progress bar.
    waitbar(p/L);
   end
end
close(bar)

G = @tableLookupFcn;

   %-----------------------------------------------------------------%
   function out = tableLookupFcn(varargin)
      if num_inputs > 1
         out = interpn(grid{:}, table, varargin{:});
      else
         out = interp1(grid{1}, table, varargin{1});
      end
   end

end
```

`interpn` is a multidimensional version of `interp1`, discussed in Section 3.2.3. See the footnote in that section regarding the use of interpolation functions to perform table lookup operations. See the help page for `interpn` for more details.



■ In this example we use the fuzzy functions to compute the percent of operational abnormality of the motor example based on the functions in Figs. 3.22–3.24. First, we demonstrate the use of the individual functions and then we obtain the solution in one step using function `fuzzysysfcn`.

EXAMPLE 3.14: Using the fuzzy functions.

We begin by generating handles for the input membership functions in Fig. 3.22:

```
>> ulow = @(z) 1 - sigmamf(z, 0.27, 0.47);
>> umid = @(z) triangmf(z, 0.24, 0.50, 0.74);
>> uhigh = @(z) sigmamf(z, 0.53, 0.73);
```

These functions correspond approximately to the plots in Fig. 3.22(a) (note how we used `1 - sigmamf` to generate the leftmost function in that figure). You can display a plot of these functions by typing:

```
>> fplot(ulow, [0 1], 20);
>> hold on
>> fplot(umid, [0 1],'-.', 20);
>> fplot(uhigh, [0 1],'--', 20);
>> hold off
>> title('Input membership functions, Example 3.14')
```

See Section 3.3.1 regarding function `fplot`.

Similarly, the following three output functions correspond approximately to the plots in Fig. 3.23.

```
>> unorm = @(z) 1 - sigmamf(z, 0.18, 0.33);
>> umarg = @(z) trapezmf(z, 0.23, 0.35, 0.53, 0.69);
>> ufail = @(z) sigmamf(z, 0.59, 0.78);
```

Next we arrange the input membership function handles in a cell array and obtain the rule strengths. Note the use of semicolons in array `rules` because `lambdafcns` expects each row of the array to contain the membership functions associated with that rule (in this case there is only one input membership function per rule):

```
>> rules = {ulow; umid; uhigh};
>> L = lambdafcns(rules);
```

To generate the results of implication we need `L`, the three output functions constructed earlier, and a specific value of `z` (which is a scalar, as there is only one input value in this case):

```
>> z = 0.7; % See Fig. 3.22(b).
>> outputmfs = {unorm, umarg, ufail};
>> Q = implfcns(L, outputmfs, z);
```

The next step is aggregation:

```
>> Qa = aggfcn(Q);
```

and the final step is fuzzification:

```
>> final_result = defuzzify(Qa, [0 1])
final_result =
    0.7619
```

which is the 76% abnormality discussed in connection with Fig. 3.24. Using function `fuzzysysfcn` yields the same result, as expected:

```
>> F = fuzzysysfcn(rules, outputmfs, [0 1]);
>> F(0.7)
ans =
    0.7619
```

Using function `approxfcn`

```
>> G = approxfcn(F, [0 1]);
>> G(0.7)
```

```
ans =
    0.7619
```

gives the same result. In fact, if we plot the two functions

```
>> fplot(F, [0 1],'k', 20) % Plot as a black line.
>> hold on
>> fplot(G,[0 1],'k:o', 20) % Plot as circles connected by
dots.
>> hold off
```

you can see in Fig. 3.26 that the two fuzzy system responses are identical for all practical purposes.

To evaluate the time advantage between the two implementations we use function `timeit` from Section 2.10.5:

As mentioned earlier, you can see here one of the advantages of using nested functions in the development of the fuzzy function set. The fact that the outputs are functions allows us to plot the complete system response function for all possible input values.

```
>> f = @() F(0.7);
>> g = @() G(0.7);
>> t1 = timeit(f);
>> t2 = timeit(g);
>> t = t1/t2
t =
    9.4361
```

so the approximation function runs almost ten times faster in this case. ■

3.6.5 Using Fuzzy Sets for Intensity Transformations

Contrast enhancement, one of the principal applications of intensity transformations, can be expressed in terms of the following rules

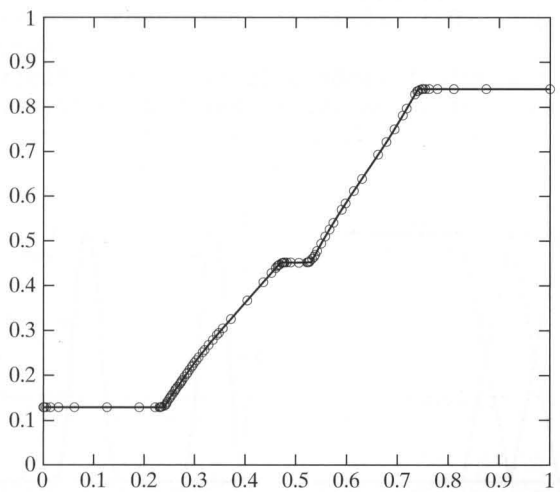


FIGURE 3.26 Comparison between the outputs of functions `fuzzysysfcn` (plotted as a solid line) and `approxfcn` (plotted as circles connected by dots). The results are visually indistinguishable. (Recall from Section 3.3.1 that `fplot` distributes the distance between plot point nonuniformly.)

IF a pixel is *dark*, THEN make it *darker*
IF a pixel is *gray*, THEN make it *gray*
IF a pixel is *bright*, THEN make it *brighter*

If we consider the terms in italics to be fuzzy, we can express the concepts of *dark*, *gray*, and *bright*, by the membership functions in Fig. 3.27(a). With respect to the output, making intensities darker and brighter means increasing the separation of dark and light on the gray scale, which increases contrast. Usually, narrowing the mid grays increases the "richness" of the image. Figure 3.27(b) shows a set of output membership functions that accomplish these objectives.

EXAMPLE 3.15: Using the fuzzy functions to implement fuzzy contrast enhancement.

■ Figure 3.28(a) shows an image, `f`, whose intensities span a narrow range of the gray scale, as the histogram in Fig. 3.29(a) (obtained using `imhist`) shows. The net result is an image with low contrast.

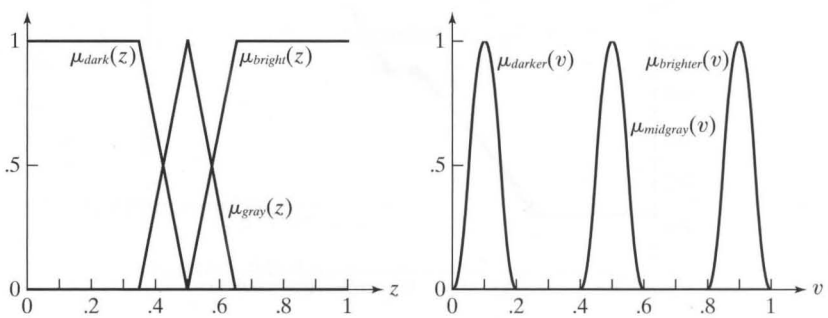
Figure 3.28(b) is the result of using histogram equalization to increase image contrast. As the histogram in Fig. 3.29(b) shows, the entire gray scale was spread out but, in this case, the spread was excessive in the sense that contrast was increased, but the result is an image with an "over exposed" appearance. For example, the details in Professor Einstein's forehead and hair are mostly lost.

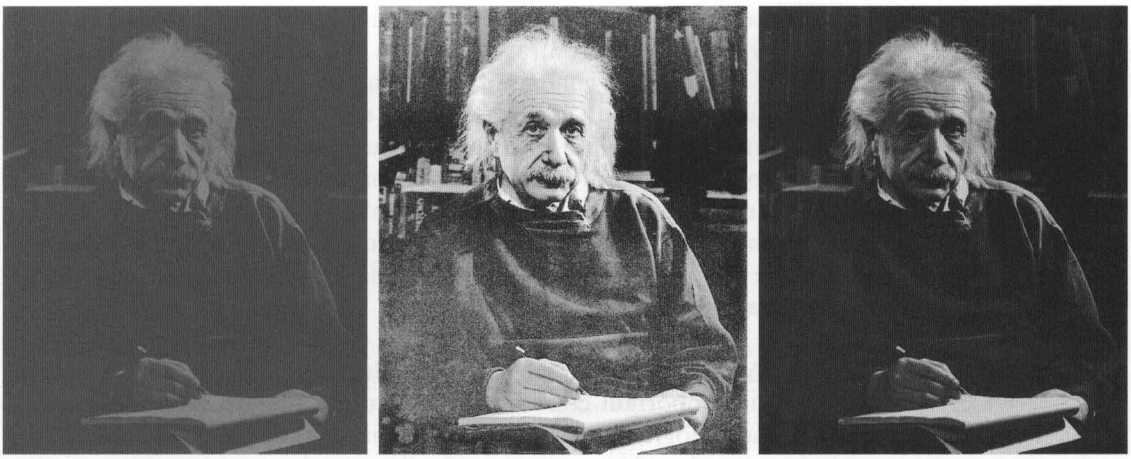
Figure 3.28(c) shows the result of the following fuzzy operations:

```
>> % Specify input membership functions
>> udark = @(z) 1 - sigmamf(z, 0.35, 0.5);
>> ugray = @(z) triangmf(z, 0.35, 0.5, 0.65);
>> ubright = @(z) sigmamf(z, 0.5, 0.65);

>> % Plot the input membership functions. See Fig. 3.27(a).
>> fplot(udark, [0 1], 20)
>> hold on
>> fplot(ugray, [0 1], 20)
>> fplot(ubright, [0 1, 20])

>> % Specify the output membership functions. Plotting of
>> % these functions is as above. See Fig. 3.27(b).
```

a b

FIGURE 3.27 (a) Input and (b) output membership functions for fuzzy, rule-based contrast enhancement.

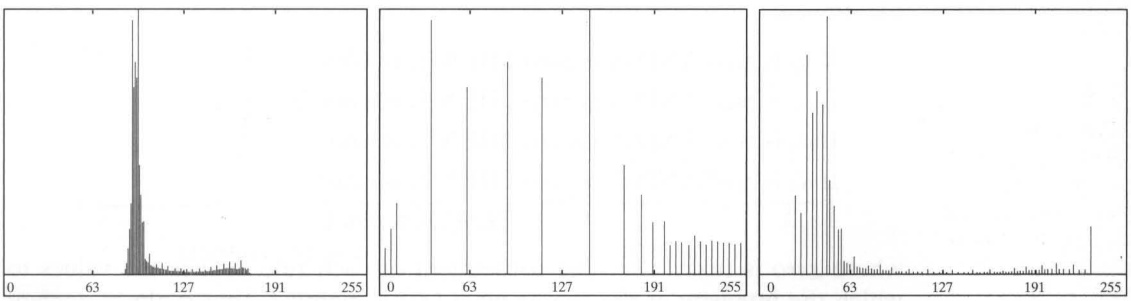




a b c

FIGURE 3.28 (a) Low-contrast image. (b) Result of histogram equalization. (c) Result of fuzzy, rule-based, contrast enhancement.

```
>> udarker = @(z) bellmf(z, 0.0, 0.1);
>> umidgray = @(z) bellmf(z, 0.4, 0.5);
>> ubrighter = @(z) bellmf(z, 0.8, 0.9);

>> % Obtain fuzzy system response function.
>> rules = {udark; ugray; ubright};
>> outmf = {udarker, umidgray, ubrighter};
>> F = fuzzysysfcn(rules, outmf, [0 1]);

>> % Use F to construct an intensity transformation function.
>> z = linspace(0, 1, 256); % f is of class uint8.
>> T = F(z);
>> % Transform the intensities of f using T.
>> g = intrans(f, 'specified', T);
>> figure, imshow(g)
```



a b c

FIGURE 3.29 Histograms of the images in Fig. 3.28(a), (b), and (c), respectively.

As you can see in Fig. 3.28(c) the result of the preceding fuzzy operations is an image having increased contrast and a rich gray tonality. Note, for example, the hair and forehead, as compared to the same regions in Fig. 3.28(b). The reason for the improvement can be explained easily by studying the histogram of Fig. 3.28(c), shown in Fig. 3.29(c). Unlike the histogram of the equalized image, this histogram has kept the same basic characteristics of the histogram of the original image. However, it is quite evident that the dark levels (tall peaks in the low end of the histogram) were moved left, thus darkening the levels. The opposite was true for bright levels. The mid grays were spread slightly, but much less than in histogram equalization.

The price of this improvement in image quality is increased processing complexity. A practical approach to follow when processing speed and image throughput are important considerations is to use fuzzy techniques to determine what the histograms of well-balanced images should look like. Then, faster techniques, such as histogram specification, can be used to achieve similar results by mapping the histograms of the input images to one or more of the "ideal" histograms determined using a fuzzy approach. ■

3.6.6 Using Fuzzy Sets for Spatial Filtering

When using fuzzy sets for spatial filtering, the basic approach is to define fuzzy neighborhood properties that "capture" the essence of what the filters are supposed to detect. Fro example, we can develop a fuzzy boundary detection (enhancement) algorithm based on the following fuzzy statement:

If a pixel belongs to a uniform region, then make it white; else make it black

where *black* and *white* are fuzzy variables. To express the concept of a "uniform region" in fuzzy terms, we can consider the intensity differences between the pixel at the center of the neighborhood and its neighbors. For the 3×3 neighborhood in Fig. 3.30(a), the differences between the center pixel (labeled z_5) and each of the neighbors form the subimage of size 3×3 in Fig. 3.30(b), where d_i denotes the intensity difference between the ith neighbor and the center point (i.e., $d_i = z_i - z_5$, where the z's are intensity values). The following four IF-THEN rules and one ELSE rule implement the fuzzy statement just mentioned:

If d_2 is *zero* AND d_6 is *zero* THEN z_5 is *white*
If d_6 is *zero* AND d_8 is *zero* THEN z_5 is *white*
If d_8 is *zero* AND d_4 is *zero* THEN z_5 is *white*
If d_4 is *zero* AND d_2 is *zero* THEN z_5 is *white*
ELSE z_5 is *black*

where *zero* is fuzzy also. The consequent of each rule defines the values to which the intensity of the center pixel (z_5) is mapped. That is, the statement "THEN z_5 is *white*" means that the intensity of the pixel located at the center of the neighborhood is mapped to *white*. These rules state that the center pixel

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Pixel neighborhood

d_1	d_2	d_3
d_4	0	d_6
d_7	d_8	d_9

Intensity differences

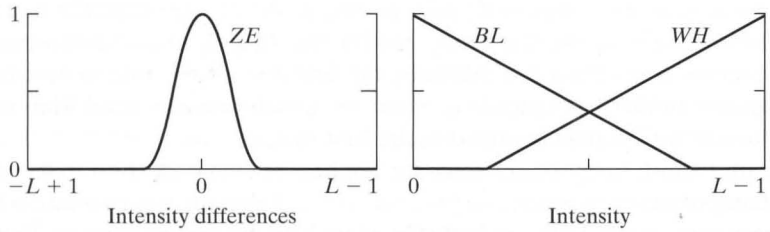
a b

FIGURE 3.30 (a) A 3×3 pixel neighborhood, and (b) corresponding intensity differences between the center pixels and its neighbors. Only d_2, d_4, d_6, and d_8 are used here to simplify the discussion.

is considered to be part of a uniform region if the intensity differences just mentioned are *zero* (in a fuzzy sense); otherwise (ELSE) it is considered a *black* (boundary) pixel.

Figure 3.31(a) shows the membership function for *zero*, which is the input membership function, and Fig. 3.31(b) shows the output membership functions *black*, and *white*, respectively, where we use ZE, BL, and WH to simplify notation. Note that the range of the independent variable of the fuzzy set ZE for an image with L possible intensity levels is $[-L + 1, L - 1]$ because intensity differences can range between $-(L - 1)$ and $L - 1$. On the other hand, the range of the output intensities is $[0, L - 1]$, as in the original image. Figure 3.32 shows graphically the rules stated above, where the box labeled z_5 indicates that the intensity of the center pixel is mapped to the output value WH or BL.

Fuzzy filtering based on the preceding concepts has two basic parts: formulation of the fuzzy filtering system, and computation of the intensity differences over an entire image. Implementation is made modular if we treat these two parts separately, which will allow changing the fuzzy approach without affecting the code that computes the differences. The approach in the following



a b

FIGURE 3.31 Membership function of the fuzzy set *zero* (ZE). (b) Membership functions of the fuzzy sets *black* (BL) and *white* (WH).

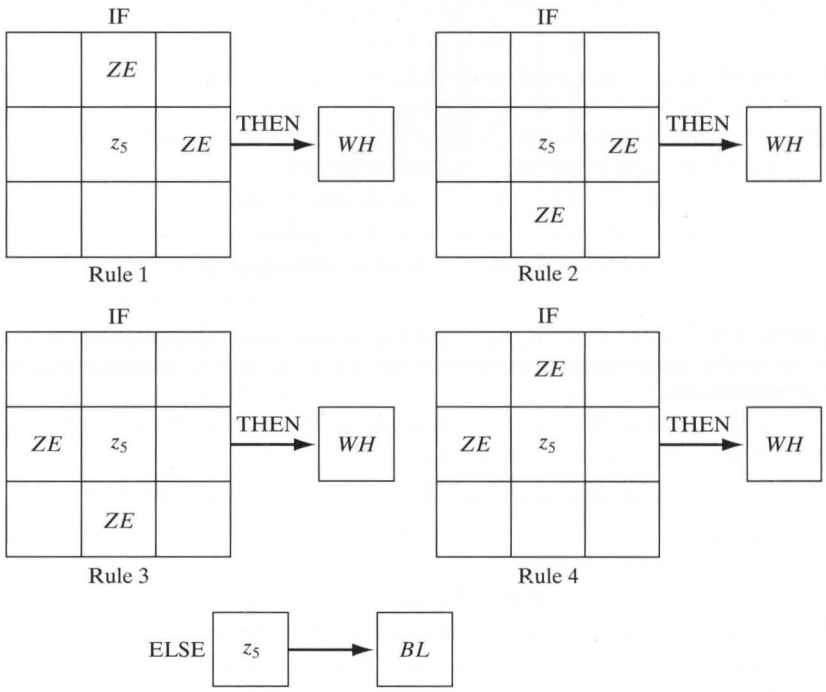


FIGURE 3.32 Fuzzy rules for boundary detection.

discussion is (1) to create a script that implements the fuzzy system and save it as a MAT-file; and (2) implement a separate filtering function that computes the differences and then loads the fuzzy system to evaluate those differences.

See Section 1.7.3 regarding saving and loading MAT-files.

We develop the script first, which we call `makefuzzyedgesys`. Note in the script code that, because not all inputs are associated with each output, we define an input rule of ones (which we call *not_used*) to designate which rules are not used for a given output (recall that we are using the min operation in the model of Fig. 3.25, so an input membership function valued 1, which is the maximum possible value, does not affect the output). Note also in the code that, because we have four rules and four inputs, the rule matrix is of size 4×4, as explained earlier in function `lambdafcns`. In the present case, the first input is d_2, the second is d_4, the third is d_6, and the fourth is d_8, and each has membership function *zero*. Then, for example, the first row of the rule matrix (which corresponds to the first output) is: *zero*, *not_used*, *zero*, *not_used*. That is, only the first and third inputs are used in the first rule.

Because these fuzzy operations are applied at every location in the image, this a computationally-intensive process, so we obtain an approximation to the fuzzy system using function `approxfcn` to reduce processing time, as discussed earlier. Because we are interested in saving only the fuzzy system approximation (called `G` in the code) in the MAT-file we use the following syntax for the `save` function:

```
save filename content
```

The rest of the code is self-explanatory.

makefuzzyedgesys

```
%MAKEFUZZYEDGESYS Script to make MAT-file used by FUZZYFILT.

% Input membership functions.
zero = @(z) bellmf(z, -0.3, 0);
not_used = @(z) onemf(z);

% Output membership functions.
black = @(z) triangmf(z, 0, 0, 0.75);
white = @(z) triangmf(z, 0.25, 1, 1);

% There are four rules and four inputs, so the inmf matrix is 4x4.
% Each row of the inmf matrix corresponds to one rule.
inmf = {zero, not_used, zero, not_used
   not_used, not_used, zero, zero
   not_used, zero, not_used, zero
   zero, zero, not_used, not_used};

% The set of output membership functions has an "extra" one, which
% means that an "else rule" will automatically be used.
outmf = {white, white, white, white, black};

% Inputs to the output membership functions are in the range [0, 1].
vrange = [0 1];

F = fuzzysysfcn(inmf, outmf, vrange);

% Compute a lookup-table-based approximation to the fuzzy system
% function. Each of the four inputs is in the range [-1, 1].
G = approxfcn(F, [-1 1; -1 1; -1 1; -1 1]);

% Save the fuzzy system approximation function to a MAT-file.
save fuzzyedgesys G
```

Observe that separating the various rows of an array by carriage returns is equivalent to using semicolons.

Implementing a function that computes the differences is straightforward. Note in particular how the computation of these differences is done using `imfilter` and also how the fuzzy system function `G` is evaluated with all the differences at once, showing the advantage of the vectorized implementation that was used in the development of the fuzzy functions. When function `load` is called with an output argument, `load` returns a structure. Therefore, the command

```
s = load(makefuzzyedges)
```

returns `s.G`, (structure `s` with a field named `G`) because the MAT-file `makefuzzyedges` was saved with content `G`, as explained earlier.

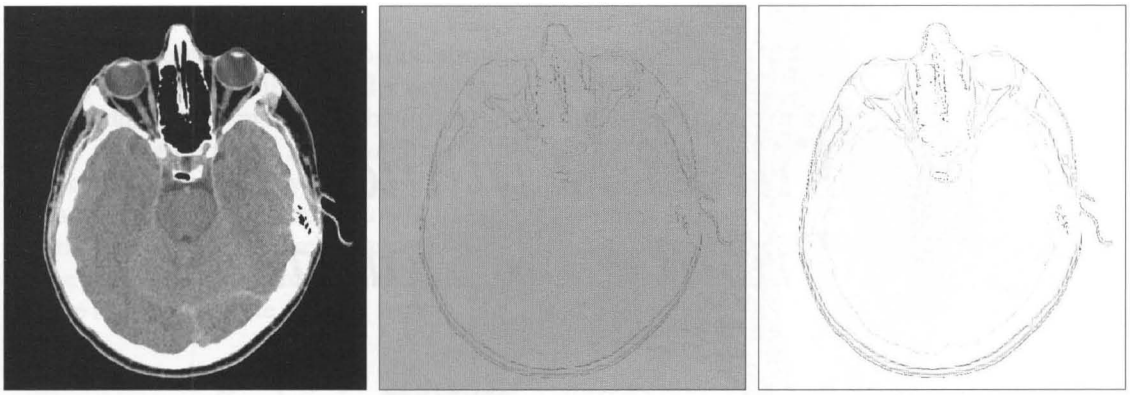
fuzzyfilt

```
function g = fuzzyfilt(f)
%FUZZYFILT Fuzzy edge detector.
%   G = FUZZYFILT(F) implements the rule-based fuzzy filter
%   discussed in the "Using Fuzzy Sets for Spatial Filtering"
%   section of Digital Image Processing Using MATLAB/2E. F and G are
%   the input and filtered images, respectively.
%
%   FUZZYFILT is implemented using precomputed fuzzy system function
%   handle saved in the MAT-file fuzzyedgesys.mat.  The M-script
%   makefuzzyedgesys.m contains the code used to create the fuzzy
%   system function.

% Work in floating point.
[f, revertClass] = tofloat(f);

% The fuzzy system function has four inputs - the differences
% between the pixel and its north, east, south, and west neighbors.
% Compute these differences for every pixel in the image using
% imfilter.
z1 = imfilter(f, [0 -1 1], 'conv', 'replicate');
z2 = imfilter(f, [0; -1; 1], 'conv', 'replicate');
z3 = imfilter(f, [1; -1; 0], 'conv', 'replicate');
z4 = imfilter(f, [1 -1 0], 'conv', 'replicate');

% Load the precomputed fuzzy system function from the MAT-file and
% apply it.
s = load('fuzzyedgesys');
g = s.G(z1, z2, z3, z4);

% Convert the output image back to the class of the input image.
g = revertClass(g);
```

EXAMPLE 3.16: Boundary detection using fuzzy, rule-based spatial filtering.

■ Figure 3.33(a) shows a 512×512 CT scan of a human head, and Fig. 3.33(b) is the result of using the fuzzy spatial filtering approach just discussed. Note the effectiveness of the method in extracting the boundaries between regions, including the contour of the brain (inner gray region).

The constant regions in the image appear gray because, when the intensity differences discussed earlier are near zero, the THEN rules have a strong response. These responses in turn clip function *WH*. The output (the center of gravity of the clipped triangular regions) is a constant between $(L-1)/2$ and $L-1$, thus producing the grayish tone seen in the image. The contrast of this image can be improved significantly by expanding the gray scale. For example, Fig. 3.33(c) was obtained by performing intensity scaling using function `mat2gray`. The net result is that the intensity values in Fig. 3.33(c) span the full gray scale. ■

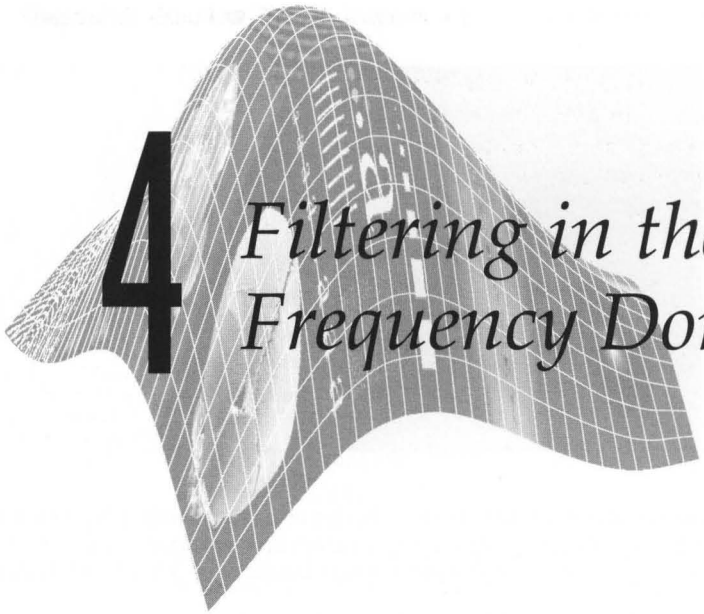


a b c

FIGURE 3.33 (a) CT scan of a human head. (b) Result of fuzzy spatial filtering using the membership functions in Fig. 3.31 and the rules in Fig. 3.32. (c) Result after intensity scaling. The thin black picture borders in (b) and (c) were added for clarity; they are not part of the data. (Original image courtesy of Dr. David R. Pickens, Vanderbilt University.)

Summary

The material in this chapter is the foundation for numerous topics that you will encounter in subsequent chapters. For example, we use spatial processing in Chapter 5 in connection with image restoration, where we also take a closer look at noise reduction and noise generating functions in MATLAB. Some of the spatial masks that were mentioned briefly here are used extensively in Chapter 11 for edge detection in segmentation applications. The concepts of convolution and correlation are explained again in Chapter 4 from the perspective of the frequency domain. Conceptually, neighborhood processing and the implementation of spatial filters will surface in various discussions throughout the book. In the process, we will extend many of the discussion begun here and introduce additional aspects of how spatial filters can be implemented efficiently in MATLAB.



4 Filtering in the Frequency Domain

Preview

For the most part, this chapter parallels the filtering topics discussed in Chapter 3, but with all filtering carried out in the frequency domain via the Fourier transform. In addition to being a cornerstone of linear filtering, the Fourier transform offers considerable flexibility in the design and implementation of filtering solutions in areas such as image enhancement, image restoration, image data compression, and a host of other applications of practical interest. In this chapter, the focus is on the foundation of how to perform frequency domain filtering in MATLAB. As in Chapter 3, we illustrate filtering in the frequency domain with examples of image enhancement, including lowpass filtering for image smoothing, highpass filtering (including high-frequency emphasis filtering) for image sharpening, and selective filtering for the removal of periodic interference. We also show briefly how spatial and frequency domain processing can be used in combination to yield results that are superior to using either type of processing alone. Although most of the examples in this chapter deal with image enhancement, the concepts and techniques developed in the following sections are quite general, as illustrated by other applications of this material in Chapters 5, 9, and 11, 12, and 13.

4.1 The 2-D Discrete Fourier Transform

Let $f(x, y)$ for $x = 0, 1, 2, \ldots, M - 1$ and $y = 0, 1, 2, \ldots, N - 1$ denote a digital image of size $M \times N$ pixels. The 2-D *discrete Fourier transform* (DFT) of $f(x, y)$, denoted by $F(u, v)$, is given by the equation

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-j2\pi(ux/M + vy/N)}$$

for $u = 0, 1, 2, \ldots, M-1$ and $v = 0, 1, 2, \ldots, N-1$. We could expand the exponential into sine and cosine functions, with the variables u and v determining their frequencies (x and y are summed out). The *frequency domain* is the coordinate system spanned by $F(u, v)$ with u and v as (frequency) variables. This is analogous to the *spatial domain* studied in Chapter 3, which is the coordinate system spanned by $f(x, y)$, with x and y as (spatial) variables. The $M \times N$ rectangular region defined by $u = 0, 1, 2, \ldots, M-1$ and $v = 0, 1, 2, \ldots, N-1$ is often referred to as the *frequency rectangle*. Clearly, the frequency rectangle is of the same size as the input image.

The inverse, discrete Fourier transform (IDFT) is given by

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) e^{j2\pi(ux/M + vy/N)}$$

The DFT and IDFT are derived starting from basic principles in Gonzalez and Woods [2008].

for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$. Thus, given $F(u, v)$, we can obtain $f(x, y)$ back by means of the IDFT. The values of $F(u, v)$ in this equation sometimes are referred to as the *Fourier coefficients* of the expansion.

In some formulations of the DFT, the $1/MN$ term appears in front of the transform and in others it is used in front of the inverse. MATLAB's implementation uses the term in front of the inverse, as in the preceding equation. Because array indices in MATLAB start at 1 rather than 0, `F(1, 1)` and `f(1, 1)` in MATLAB correspond to the mathematical quantities $F(0, 0)$ and $f(0, 0)$ in the transform and its inverse. In general `F(i, j)` $= F(i-1, j-1)$ and `f(i, j)` $= f(i-1, j-1)$ for $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$.

The value of the transform at the origin of the frequency domain [i.e., $F(0, 0)$] is called the *dc component* of the Fourier transform. This terminology is from electrical engineering, where "dc" signifies direct current (current of zero frequency). It is not difficult to show that $F(0, 0)$ is equal to MN times the average value of $f(x, y)$.

Even if $f(x, y)$ is a real function, its transform is complex in general. The principal method for analyzing a transform visually is to compute its *spectrum* [i.e., the magnitude of $F(u, v)$, which is a real function] and display it as an image. Letting $R(u, v)$ and $I(u, v)$ represent the real and imaginary components of $F(u, v)$, the *Fourier spectrum* is defined as

$$|F(u, v)| = \left[R^2(u, v) + I^2(u, v)\right]^{\frac{1}{2}}$$

The *phase angle* of the transform is defined as

$$\phi(u, v) = \arctan\left[\frac{I(u, v)}{R(u, v)}\right]$$

Because R and I can be positive and negative independently, the arctan is understood to be a *four-quadrant arctangent* (see Section 4.2).

These two functions can be used to express the complex function $F(u, v)$ in polar form:

$$F(u, v) = |F(u, v)| e^{j\phi(u, v)}$$

The *power spectrum* is defined as the square of the magnitude:

$$\begin{aligned} P(u,v) &= |F(u,v)|^2 \\ &= R^2(u,v) + I^2(u,v) \end{aligned}$$

For purposes of visualization it typically is immaterial whether we view $|F(u,v)|$ or $P(u,v)$.

If $f(x,y)$ is real, its Fourier transform is conjugate symmetric about the origin; that is,

$$F(u,v) = F^*(-u,-v)$$

This implies that the Fourier spectrum is symmetric about the origin also:

$$|F(u,v)| = |F(-u,-v)|$$

It can be shown by direct substitution into the equation for $F(u,v)$ that

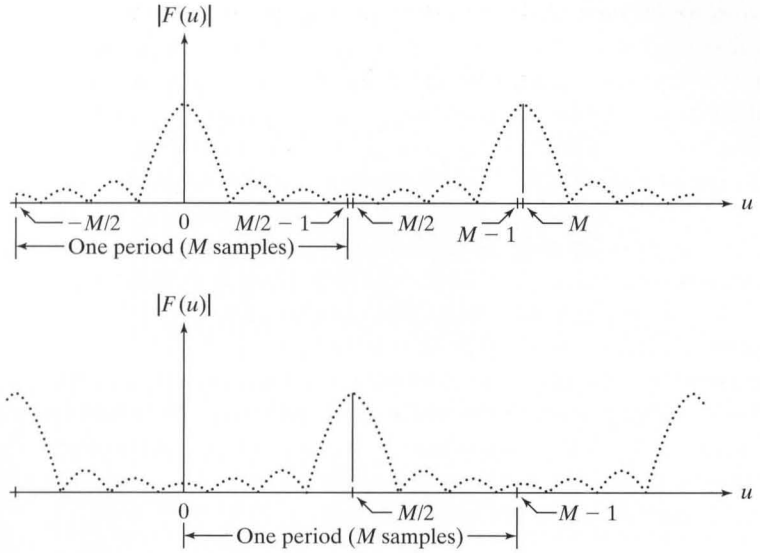
$$F(u,v) = F(u + k_1M, v) = F(u, v + k_2N) = F(u + k_1M, v + k_2N)$$

where k_1 and k_2 are integers. In other words, the DFT is infinitely periodic in both the u and v directions, with the periodicity determined by M and N. Periodicity is a property of the inverse DFT also:

$$f(x,y) = f(x + k_1M, y) = f(x, y + k_2N) = f(x + k_1M, y + k_2N)$$

That is, an image obtained by taking the inverse Fourier transform is also infinitely periodic. This is a frequent source of confusion because it is not at all intuitive why images resulting from taking the inverse Fourier transform should be periodic. It helps to remember that this is simply a mathematical property of the DFT and its inverse. Keep in mind also that DFT implementations compute only one period, so we work with arrays of size $M \times N$.

The periodicity issue becomes important when we consider how DFT data relate to the periods of the transform. For instance, Fig. 4.1(a) shows the spectrum of a one-dimensional transform, $F(u)$. In this case, the periodicity expression becomes $F(u) = F(u + k_1M)$, from which it follows that $|F(u)| = |F(u + k_1M)|$. Also, because of symmetry, $|F(u)| = |F(-u)|$. The periodicity property indicates that $F(u)$ has a period of length M, and the symmetry property indicates that $|F(u)|$ is centered on the origin, as Fig. 4.1(a) shows. This figure and the preceding comments demonstrate that the values of $|F(u)|$ from $M/2$ to $M-1$ are repetitions of the values in the half period to the left of the origin. Because the 1-D DFT is implemented for only M points (i.e., for integer values of u in the interval $[0, M-1]$), it follows that computing the 1-D transform yields two back-to-back half periods in this interval. We are interested in obtaining one full, properly ordered period in the interval $[0, M-1]$. It is not difficult to show (Gonzalez and Woods [2008]) that the desired period is obtained by multiply-

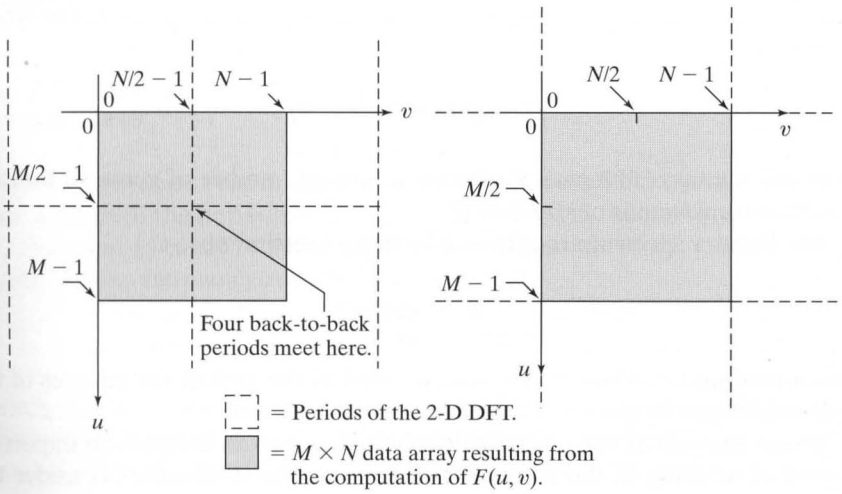


a
b

FIGURE4.1
(a) Fourier spectrum showing back-to-back half periods in the interval $[0, M-1]$. (b) Centered spectrum in the same interval, obtained by multiplying $f(x)$ by $(-1)^x$ prior to computing the Fourier transform.

ing $f(x)$ by $(-1)^x$ prior to computing the transform. Basically, what this does is move the origin of the transform to the point $u = M/2$, as Fig. 4.1(b) shows. You can see that the value of the spectrum at $u = 0$ in Fig. 4.1(b) corresponds to $|F(-M/2)|$ in Fig. 4.1(a). Similarly, the values at $|F(M/2)|$ and $|F(M-1)|$ in Fig. 4.1(b) correspond to $|F(0)|$ and $|F(M/2-1)|$ in Fig. 4.1(a).

A similar situation exists with two-dimensional functions. Computing the 2-D DFT now yields transform points in the rectangular interval shown in Fig. 4.2(a), where the shaded area indicates values of $F(u, v)$ obtained by implementing the 2-D Fourier transform equation defined at the beginning of this section. The



a b

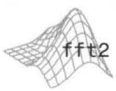
FIGURE 4.2
(a) $M \times N$ Fourier spectrum (shaded), showing four back-to-back quarter periods. (b) Spectrum after multiplying $f(x, y)$ by $(-1)^{x+y}$ prior to computing the Fourier transform. The shaded period is the data that would be obtained by using the DFT.

dashed rectangles are periodic repetitions, as in Fig. 4.1(a). The shaded region shows that the values of $F(u, v)$ now encompass four back-to-back quarter periods that meet at the point shown in Fig. 4.2(a). Visual analysis of the spectrum is simplified by moving the values at the origin of the transform to the center of the frequency rectangle. This can be accomplished by multiplying $f(x, y)$ by $(-1)^{x+y}$ prior to computing the 2-D Fourier transform. The periods then would align as in Fig. 4.2(b). The value of the spectrum at coordinates $(M/2, N/2)$ in Fig. 4.2(b) is the same as its value at $(0, 0)$ in Fig. 4.2(a), and the value at $(0, 0)$ in Fig. 4.2(b) is the same as the value at $(-M/2, -N/2)$ in Fig. 4.2(a). Similarly, the value at $(M - 1, N - 1)$ in Fig. 4.2(b) is the same as the value at $(M/2 - 1, N/2 - 1)$ in Fig. 4.2(a).

The preceding discussion for centering the transform by multiplying $f(x, y)$ by $(-1)^{x+y}$ is an important concept that is included here for completeness. When working in MATLAB, the approach is to compute the transform without multiplication by $(-1)^{x+y}$ and then to rearrange the data afterwards using function `fftshift`, discussed in the following section.

4.2 Computing and Visualizing the 2-D DFT in MATLAB

The DFT and its inverse are obtained in practice using a fast Fourier transform (FFT) algorithm. The FFT of an image array `f` is obtained in MATLAB using function `fft2`, which has the syntax:




```
F = fft2(f)
```

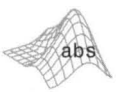
This function returns a Fourier transform that is also of size $M \times N$, with the data arranged in the form shown in Fig. 4.2(a); that is, with the origin of the data at the top left, and with four quarter periods meeting at the center of the frequency rectangle.

As explained in Section 4.3.1, it is necessary to pad the input image with zeros when the Fourier transform is used for filtering. In this case, the syntax becomes

```
F = fft2(f, P, Q)
```

With this syntax, `fft2` pads `f` with the required number of zeros so that the resulting transform is of size $P \times Q$.

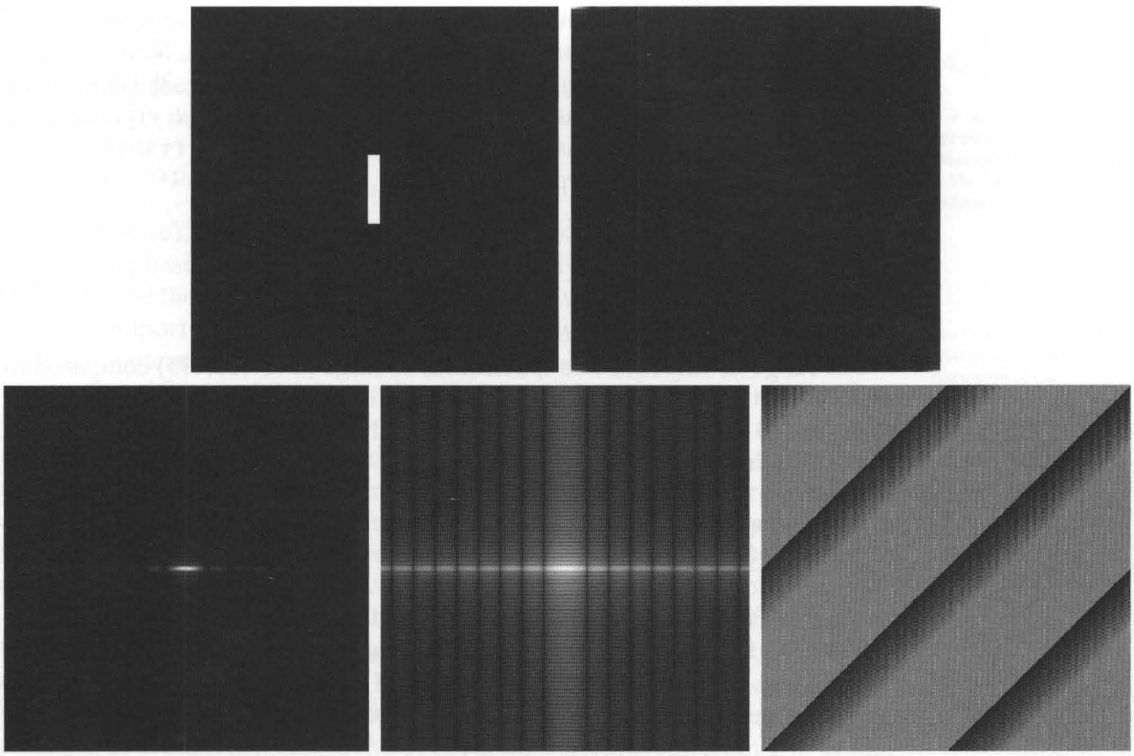
The Fourier spectrum is obtained by using function `abs`:




```
S = abs(F)
```

which computes the magnitude (square root of the sum of the squares of the real and imaginary parts) of each element of the array.

Visual analysis of the spectrum by displaying it as an image is an important aspect of working in the frequency domain. As an illustration, consider the



a b
c d e

FIGURE 4.3 (a) Image. (b) Fourier spectrum. (c) Centered spectrum. (d) Spectrum visually enhanced by a log transformation. (e) Phase angle image.

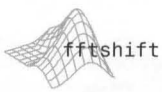
image, `f`, in Fig. 4.3(a). We compute its Fourier transform and display the spectrum using the following commands:

```
>> F = fft2(f);
>> S = abs(F);
>> imshow(S, [])
```

Figure 4.3(b) shows the result. The four bright spots in the corners of the image are a result of the periodicity property mentioned in the previous section.

Function `fftshift` can be used to move the origin of the transform to the center of the frequency rectangle. The syntax is

```
Fc = fftshift(F)
```



where `F` is the transform computed using `fft2` and `Fc` is the centered transform. Function `fftshift` operates by swapping the quadrants of `F`. For example, if `a = [1 2; 3 4]`, then `fftshift(a) = [4 3; 2 1]`. When applied to a Fourier

transform, the net result of using `fftshift` is the same as if the input image had been multiplied by $(-1)^{x+y}$ prior to computing the transform. Note, however, that the two processes are *not* interchangeable. That is, letting $\Im[\cdot]$ denote the Fourier transform of the argument, we have that $\Im[(-1)^{x+y} f(x, y)]$ is equal to `fftshift(fft2(f))`, but this quantity is not equal to `fft2(fftshift(f))`.

In the present example, typing

```
>> Fc = fftshift(F);
>> imshow(abs(Fc), [ ])
```

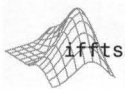
yielded the result in Fig. 4.3(c), where centering is evident.

The range of values in this spectrum is so large (0 to 420,495) compared to the 8 bits of the display that the bright values in the center dominate the result. As discussed in Section 3.2.2, this difficulty is handled via a log transformation. Thus, the commands

```
>> S2 = log(1 + abs(Fc));
>> imshow(S2, [ ])
```

resulted in Fig. 4.3(d). The increase in visual detail is significant.

Function `ifftshift` reverses the centering. Its syntax is




```
F = ifftshift(Fc)
```

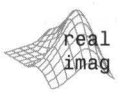
This function can be used also to convert a function that is initially centered on a rectangle to a function whose center is at the top, left corner of the rectangle. We use this property in Section 4.4.

Next we consider computation of the phase angle. With reference to the discussion in the previous section, the real and imaginary components of the 2-D Fourier transform, $R(u, v)$ and $I(u, v)$, respectively, are arrays of the same size as $F(u, v)$. Because the elements of R and I can be positive and negative independently, we need to be able to compute the arctangent in the full $[-\pi, \pi]$ range (functions with this property are called *four-quadrant arctangents*). MATLAB's function `atan2` performs this computation. Its syntax is




```
phi = atan2(I, R)
```

where `phi` is an array of the same size as `I` and `R`. The elements of `phi` are angles in radians in the range $[-\pi, \pi]$ measured with respect to the real axis. For example, `atan2(1, 1)`, `atan2(1, -1)`, and `atan2(-1, -1)` are 0.7854, 2.3562, and −2.3562 radians, or 45°, 135°, and −135°, respectively. In practice, we would write the preceding expression as

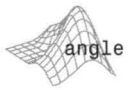



```
>> phi = atan2(imag(F), real(F));
```

`real(arg)` and `imag(arg)` extract the real and imaginary parts of `arg`, respectively.

Instead of extracting the real and imaginary components of `F`, we can use function `angle` directly:

```
phi = angle(F)
```



`P = angle(Z)` returns the phase angle of each element of complex array `Z`. The angles are in radians, in the range $\pm\pi$.

The result is the same. Given the spectrum and its corresponding phase angle, we can obtain the DFT using the expression

```
>> F = S.*exp(i*phi);
```

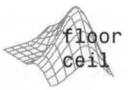
Figure 4.3(e) shows array `phi` for the DFT of Fig. 4.3(a), displayed as an image. The phase angle is not used for visual analysis as frequently as the spectrum because the former quantity is not as intuitive. However, the phase angle is just as important in terms of information content. The components of the spectrum determine the amplitude of the sinusoids that combine to form an image. The phase carries information about the displacement of various sinusoids with respect to their origin. Thus, while the spectrum is an array whose components determine the intensities of an image, the corresponding phase is an array of angles that carry information about where objects are located in an image. For example, if you displace the rectangle from the position shown in Fig. 4.3(a), its spectrum will be identical to the spectrum in Fig. 4.3(b); the displacement of the object would be reflected as a change in the phase angle.

See Gonzalez and Woods [2008] for a detailed discussion of the properties of, and interrelationship between, the spectrum and phase angle of the Fourier transform.

Before leaving the subject of the DFT and its centering, keep in mind that the center of the frequency rectangle is at $(M/2, N/2)$ if the variables u and v range from 0 to $M-1$ and $N-1$, respectively. For example, the center of an 8×8 frequency square is at point $(4,4)$ which is the 5th point along each axis, counting up from $(0,0)$. If, as in MATLAB, the variables run from 1 to M and 1 to N, respectively, then the center of the square is at $(M/2+1, N/2+1)$. That is, in this example, the center would be at point $(5,5)$, counting up from $(1,1)$. Obviously, the two centers are the same point, but this can be a source of confusion when deciding how to specify the location of DFT centers in MATLAB computations.

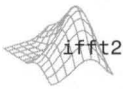
If M and N are odd, the center for MATLAB computations is obtained by rounding $M/2$ and $N/2$ down to the closest integer. The rest of the analysis is as in the previous paragraph. For example, the center of a 7×7 region is at $(3,3)$ if we count up from $(0,0)$ and at $(4,4)$ if we count up from $(1,1)$. In either case, the center is the fourth point from the origin. If only one of the dimensions is odd, the center along that dimension is similarly obtained by rounding down in the manner just explained. Using function `floor`, and keeping in mind that the MATLAB origin is at $(1,1)$, the center of the frequency rectangle for MATLAB computations is at

```
[floor(M/2) + 1, floor(N/2) + 1]
```



`B = floor(A)` rounds each element of `A` to the nearest integer less than or equal to its value. Function `ceil` rounds each element of `A` to the nearest integer greater than or equal its value.

The center given by this expression is valid both for odd and even values of `M` and `N`. In this context, a simple way to remember the difference between functions `fftshift` and `ifftshift` discussed earlier is that the former rearranges the data so that the value at location $(1,1)$ is moved to the center of the frequency rectangle, while `ifftshift` rearranges the data so that the value at the center of the frequency rectangle is moved to location $(1,1)$.



Finally, we point out that the inverse Fourier transform is computed using function `ifft2`, which has the basic syntax

```
f = ifft2(F)
```

where `F` is the Fourier transform and `f` is the resulting image. Because `fft2` converts the input image to class `double` *without scaling*, care has to be exercised in interpreting the results of the inverse. For example, if `f` is of class `uint8` its values are integers in the range [0 255], and `fft2` converts it to class `double` in the same range. Therefore, the result of the operation `ifft2(F)`, which in theory should be the same as `f`, is an image with values in the same [0 255] range, but of class `double` instead. This change in image class can lead to difficulties if not accounted for properly. Because most of our applications of `fft2` involve at some point the use of `ifft2` to get back to the spatial domain, the procedure we follow in the book is to use function `tofloat` to convert input images to floating point in the range [0 1] and then, at the end of the procedure, we use the `revertclass` feature of `tofloat` to convert the result to the same class as the original. This way, we do not have to be concerned with scaling issues.

See Section 2.7 for a discussion of function `tofloat`.

If the input image used to compute `F` is real, the inverse in theory should be real. In earlier versions of MATLAB, however, the output of `ifft2` often has small imaginary components resulting from round-off errors in computation, and common practice is to extract the real part of the result after computing the inverse to obtain an image consisting only of real values. The two operations can be combined:

```
>> f = real(ifft2(F));
```

Staring with MATLAB 7, `ifft2` performs a check to see if its input is conjugate symmetric. It it is, `ifft2` outputs a real result. Conjugate symmetry is applicable to all the work in this chapter and, because we use MATLAB 7 in the book, we do not perform the preceding operation. However, you should be aware of this issue in situations where older versions of MATLAB may be in use. This feature in MATLAB 7 is a good check on the correctness of filters. If you are working as we do in this book with a real image and symmetric, real filters, a warning from MATLAB 7 that imaginary parts are present in the result is an indication that something is incorrect either in the filter or in the procedure you are using to apply it.

Finally, note that, if padding was used in the computation of the transform, the image resulting from FFT computations is of size $P \times Q$, whereas the original image was of size $M \times N$. Therefore, results must be cropped to this original size. The procedure for doing this is discussed in the next section.

4.3 Filtering in the Frequency Domain

In this section we give a brief overview of the concepts involved in frequency domain filtering and its implementation in MATLAB.

4.3.1 Fundamentals

The foundation for linear filtering in both the spatial and frequency domains is the *convolution theorem*, which may be written symbolically as

$$f(x, y) \star h(x, y) \Leftrightarrow H(u, v)F(u, v)$$

and, conversely,

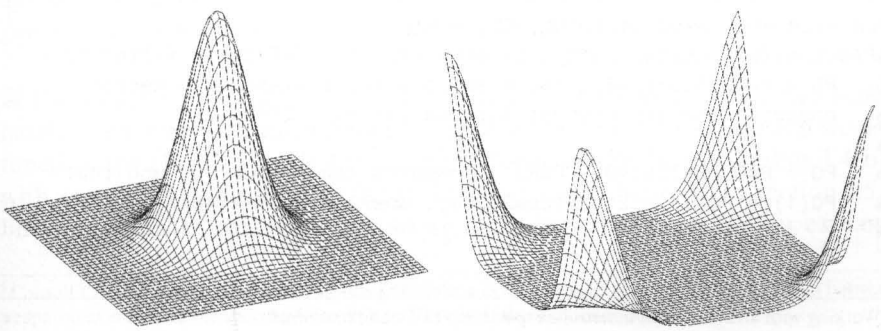
$$f(x, y)h(x, y) \Leftrightarrow H(u, v) \star F(u, v)$$

Convolution is commutative, so the order of the multiplication is immaterial. For a detailed discussion of convolution and its properties, consult Gonzalez and Woods [2008].

The symbol "★" indicates convolution of the two functions, and the expressions on the sides of the double arrow constitute a Fourier transform pair. For example, the first expression indicates that convolution of two spatial functions (the term on the left side of the expression) can be obtained by computing the inverse Fourier transform of the product of the Fourier transforms of the two functions (the term on the right side of the expression). Conversely, the forward Fourier transform of the convolution of two spatial functions gives the product of the transforms of the two functions. Similar comments apply to the second expression. In terms of filtering, we are interested in the first expression.

For reasons that will become clear shortly, function $H(u, v)$ is referred to as a *filter transfer function*, and the idea in frequency domain filtering is to select a filter transfer function that modifies $F(u, v)$ in a specified manner. For example, the filter in Fig. 4.4(a) has a transfer function that, when multiplied by a centered $F(u, v)$, attenuates the high-frequency components of $F(u, v)$, while leaving the low frequencies relatively unchanged. Filters with this characteristic are called *lowpass filters*. As discussed in Section 4.5.2, the net result of lowpass filtering is image blurring (smoothing). Figure 4.4(b) shows the same filter after it was processed with `fftshift`. This is the filter format used most frequently in the book when dealing with frequency domain filtering in which the Fourier transform of the input is not centered.

As explained in Section 3.4.1, filtering in the spatial domain consists of convolving an image $f(x, y)$ with a filter mask, $h(x, y)$. The functions are displaced with respect to each other until one of the functions slides completely past the other. According to the convolution theorem, we should get the same result in the frequency domain by multiplying $F(u, v)$ by $H(u, v)$, the Fourier transform of the spatial filter. However, when working with discrete quantities we



a b

FIGURE 4.4 Transfer functions of (a) a centered lowpass filter, and (b) the format used for DFT filtering. Note that these are frequency domain filters.

know that F and H are periodic, which implies that convolution performed in the discrete frequency domain is periodic also. For this reason, convolution performed using the DFT is called *circular convolution*. The only way to guarantee that spatial and circular convolution give the same result is to use appropriate zero-padding, as explained in the following paragraph.

Based on the convolution theorem, we know that to obtain the corresponding filtered image in the spatial domain we compute the inverse Fourier transform of the product $H(u,v)F(u,v)$. As we just explained, images and their transforms are periodic when working with DFTs. It is not difficult to visualize that convolving periodic functions can cause interference between adjacent periods if the periods are close with respect to the duration of the nonzero parts of the functions. This interference, called *wraparound error*, can be avoided by padding the functions with zeros, in the following manner.

Assume that functions $f(x,y)$ and $h(x,y)$ are of size $A \times B$ and $C \times D$ respectively. We form two *extended* (*padded*) functions, both of size $P \times Q$, by appending zeros to f and g. It can be shown (Gonzalez and Woods [2008]) that wraparound error is avoided by choosing

$$P \geq A + C - 1$$

and

$$Q \geq B + D - 1$$

Most of the work in this chapter deals with functions of the same size, $M \times N$, in which case we use the following padding values: $P \geq 2M - 1$ and $Q \geq 2N - 1$.

The following function, called `paddedsize`, computes the minimum even[†] values of P and Q required to satisfy the preceding equations. The function also has an option to pad the inputs to form square images of size equal to the nearest integer power of 2. Execution time of FFT algorithms depends roughly on the number of prime factors in P and Q. These algorithms generally are faster when P and Q are powers of 2 than when P and Q are prime. In practice, it is advisable to work with square images and filters so that filtering is the same in both directions. Function `paddedsize` provides the flexibility to do this via the choice of the input parameters. In the following code, the vectors `AB`, `CD`, and `PQ` have elements `[A B]`, `[C D]`, and `[P Q]`, respectively, where these quantities are as defined above.

`paddedsize`

```
function PQ = paddedsize(AB, CD, PARAM)
%PADDEDSIZE Computes padded sizes useful for FFT-based filtering.
%   PQ = PADDEDSIZE(AB), where AB is a two-element size vector,
%   computes the two-element size vector PQ = 2*AB.
%
%   PQ = PADDEDSIZE(AB, 'PWR2') computes the vector PQ such that
%   PQ(1) = PQ(2) = 2^nextpow2(2*m), where m is MAX(AB).
%
```

[†] Working with arrays of even dimensions speeds-up FFT computations.

```
%   PQ = PADDEDSIZE(AB, CD), where AB and CD are two-element size
%   vectors, computes the two-element size vector PQ.  The elements
%   of PQ are the smallest even integers greater than or equal to
%   AB + CD - 1.
%
%   PQ = PADDEDSIZE(AB, CD, 'PWR2') computes the vector PQ such that
%   PQ(1) = PQ(2) = 2^nextpow2(2*m), where m is MAX([AB CD]).

if nargin == 1
   PQ  = 2*AB;
elseif nargin == 2 && ~ischar(CD)
   PQ = AB + CD - 1;
   PQ = 2 * ceil(PQ / 2);
elseif nargin == 2
   m = max(AB); % Maximum dimension.

   % Find power-of-2 at least twice m.
   P = 2^nextpow2(2*m);
   PQ = [P, P];
elseif (nargin == 3) && strcmpi(PARAM, 'pwr2')
   m = max([AB CD]); % Maximum dimension.
   P = 2^nextpow2(2*m);
   PQ = [P, P];
else
   error('Wrong number of inputs.')
end
```



`p = nextpow2(n)` returns the smallest integer power of 2 that is greater than or equal to the absolute value of n.

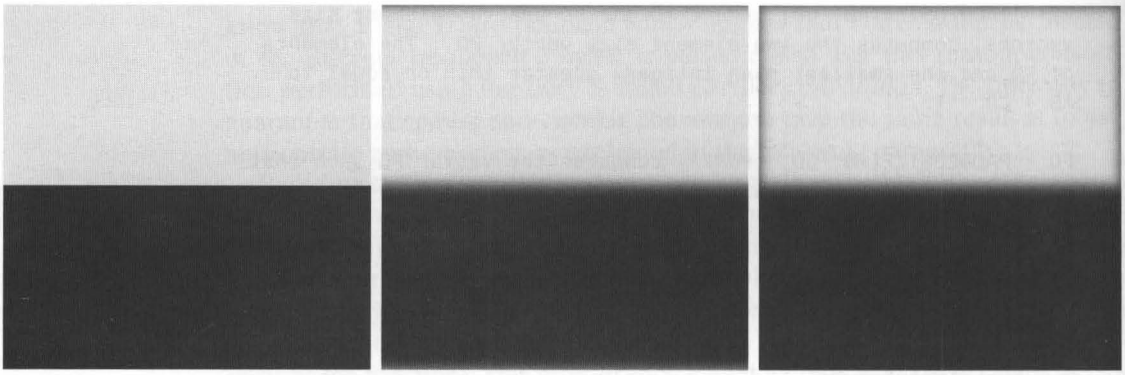
This syntax appends enough zeros to `f` such that the resulting image is of size `PQ(1)` × `PQ(2)`. Note that when `f` is padded, the filter function in the frequency domain must be of size `PQ(1)` × `PQ(2)` also.

We mentioned earlier in this section that the discrete version of the convolution theorem requires that both functions being convolved be padded in the spatial domain. This is required to avoid wraparound error. When filtering, one of the two functions involved in convolution is the filter. However, in frequency domain filtering using the DFT we specify the filter *directly* in the frequency domain, and of a size equal to the padded image. In other words, we do not pad the filter in the spatial domain.[†] As a result, it cannot be guaranteed that wraparound error is eliminated completely. Fortunately, the padding of the image, combined with the smooth shape of the filters in which we are interested generally results in negligible wraparound error.

EXAMPLE 4.1: Effects of filtering with and without padding.

■ The image, `f`, in Fig. 4.5(a) is used in this example to illustrate the difference between filtering with and without padding. In the following discussion we use function `lpfilter` to generate a Gaussian lowpass filter [similar to Fig. 4.4(b)] with a specified value of sigma (`sig`). This function is discussed in Section 4.5.2, but the syntax is straightforward, so we use it here and defer further explana-

[†] Consult Chapter 4 in Gonzalez and Woods [2008] for a detailed explanation of the relationship between wraparound error and the specification of filters directly in the frequency domain.



a b c

FIGURE 4.5 (a) An image of size 256×256 pixels. (b) Image lowpass-filtered in the frequency domain without padding. (c) Image lowpass-filtered in the frequency domain with padding. Compare the upper portion of the vertical edges in (b) and (c).

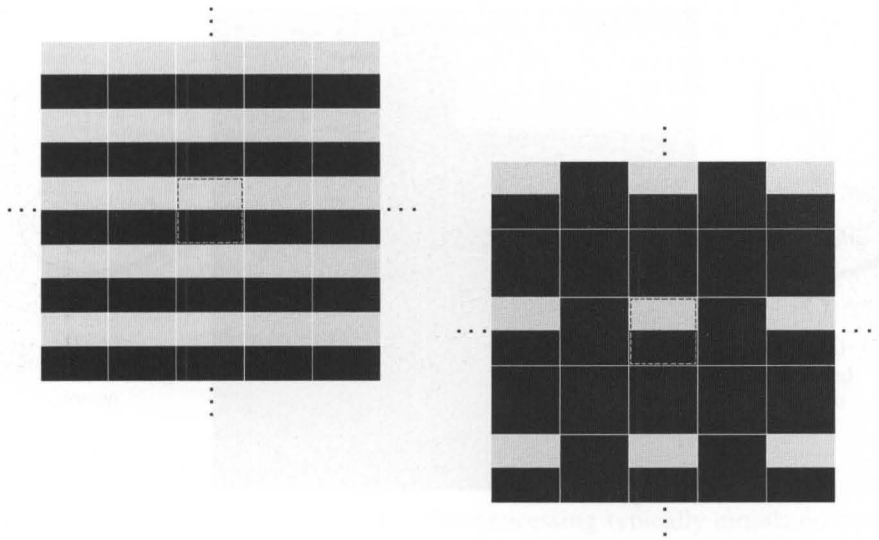
tion of `lpfilter` to that section.

The following commands perform filtering without padding:

```
>> [M, N] = size(f);
>> [f, revertclass] = tofloat(f);
>> F = fft2(f);
>> sig = 10;
>> H = lpfilter('gaussian', M, N, sig);
>> G = H.*F;
>> g = ifft2(G);
>> g = revertclass(g);
>> imshow(g)
```

Note the use of function `tofloat` to convert the input to floating point and thus avoid scaling issues with `fft2`, as explained at the end of Section 4.2. The output is converted back to the same class as the input using `revertclass`, as explained in Section 2.7. If the input image is not already floating point, `tofloat` converts it to class `single`. Frequency domain processing is memory-intensive and working whenever possible with `single`, rather than `double`, floating point helps reduce memory usage significantly.

Figure 4.5(b) shows image `g`. As expected, the image is blurred, but note that the vertical edges are not. The reason can be explained with the aid of Fig. 4.6(a), which shows graphically the implied periodicity in DFT computations. The thin white lines between the images are included for convenience in viewing; they are not part of the data. The dashed lines are used to designate the $M \times N$ image processed by `fft2`. Imagine convolving a blurring filter with this infinite periodic sequence. It is clear that when the filter is passing through the top of the dashed image it will encompass part of the image itself and also the bottom part of the periodic component immediately above it. Thus, when a light and a dark region reside under the filter, the result will be a mid-gray, blurred output. This is precisely what the top of the image in Fig. 4.5(b) shows. On the other hand, when the filter is on a side of the dashed image, it will encounter an identical region in the periodic component adjacent to the side. Because the average of a constant region is the same constant, there is no blurring in this part of the result. Other parts of the image in Fig. 4.5(b) are explained in a similar manner.



a b

FIGURE 4.6
(a) Implied, infinite periodic sequence of the image in Fig. 4.5(a). The dashed region represents the data processed by `fft2`. (b) The same periodic sequence after padding with 0s. The thin, solid white lines in both images are shown for convenience in viewing; they are not part of the data.

Consider now filtering with padding:

```
>> PQ = paddedsize(size(f)); % f is floating point.
>> Fp = fft2(f, PQ(1), PQ(2)); % Compute the FFT with padding.
>> Hp = lpfilter('gaussian', PQ(1), PQ(2), 2*sig);
>> Gp = Hp.*Fp;
>> gp = ifft2(Gp);
>> gpc = gp(1:size(f,1), 1:size(f,2));
>> gpc = revertclass(gpc);
>> imshow(gp)
```

where we used `2*sig` because the filter size is now twice the size of the filter used without padding.

Figure 4.7 shows the full, padded result, `gp`. The final result in Fig. 4.5(c) was obtained by cropping Fig. 4.7 to the original image size (see the sixth command in the preceding code). This result can be explained with the aid of Fig. 4.6(b), which shows the dashed image padded with zeros (black) as it would be set up in `fft2(f, PQ(1), PQ(2))` prior to computing the DFT. The implied periodicity is as explained earlier. The image now has a uniform black border all around it, so convolving a smoothing filter with this infinite sequence would show a gray blur in all light edges of the images. A similar result would be obtained by performing the following *spatial* filtering,

```
>> h = fspecial('gaussian', 15, 7);
>> gs = imfilter(f, h);
```

Recall from Section 3.4.1 that this call to function `imfilter` pads the border of the image with 0s by default. ■

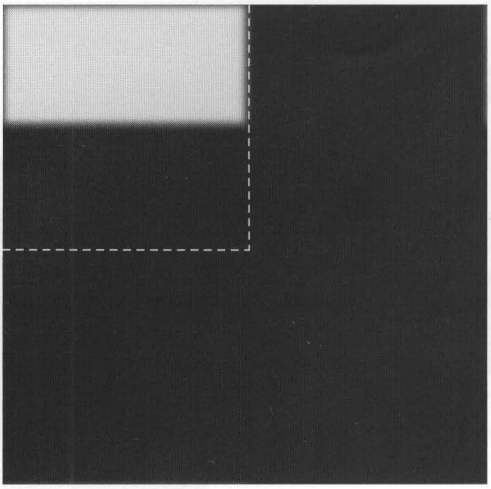


FIGURE 4.7 Full padded image resulting from `ifft2` after filtering. This image is of size 512 × 512 pixels. The dashed line shows the dimensions of the original, 256 × 256 image.

4.3.2 Basic Steps in DFT Filtering

The discussion in the previous section is summarized in the following step-by-step procedure, where `f` is the image to be filtered, `g` is the result, and it is assumed that the filter function, `H`, is of the same size as the padded image:

1. Convert the input image to floating point using function `tofloat`:
   ```
   [f, revertclass] = tofloat(f);
   ```
2. Obtain the padding parameters using function `paddedsize`:
   ```
   PQ = paddedzsize(size(f));
   ```
3. Obtain the Fourier transform with padding:
   ```
   F = fft2(f, PQ(1), PQ(2));
   ```
4. Generate a filter function, `H`, of size `PQ(1)` × `PQ(2)` using any of the methods discussed in the remainder of this chapter. The filter must be in the format shown in Fig. 4.4(b). If it is centered instead, as in Fig. 4.4(a), let `H = ifftshift(H)` before using the filter.
5. Multiply the transform by the filter:
   ```
   G = H.*F;
   ```
6. Obtain the inverse FFT of `G`:
   ```
   g = ifft2(G);
   ```
7. Crop the top, left rectangle to the original size:
   ```
   g = g(1:size(f, 1), 1:size(f, 2));
   ```
8. Convert the filtered image to the class of the input image, if so desired:
   ```
   g = revertclass(g);
   ```

Figure 4.8 shows the filtering procedure schematically. The preprocessing stage encompasses tasks such as determining image size, obtaining the padding

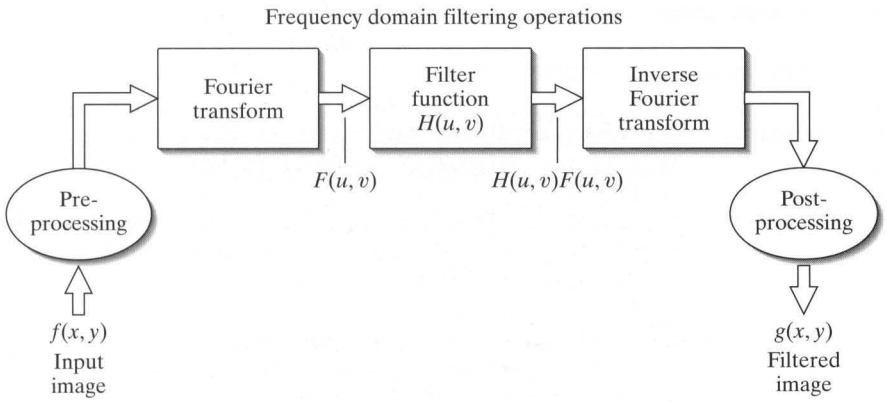


FIGURE 4.8 Basic steps for filtering in the frequency domain.

parameters, and generating a filter. Postprocessing typically entails cropping the output image and converting it to the class of the input.

The filter function $H(u, v)$ in Fig. 4.8 multiplies both the real and imaginary parts of $F(u, v)$. If $H(u, v)$ is real, then the phase of the result is not changed, a fact that can be seen in the phase equation (Section 4.1) by noting that, if the multipliers of the real and imaginary parts are equal, they cancel out, leaving the phase angle unchanged. Filters that operate in this manner are called *zero-phase-shift filters*. These are the only types of linear filters considered in this chapter.

It is well known from linear system theory that, under certain mild conditions, inputting an impulse into a linear system completely characterizes the system. When using the techniques developed in this chapter, the response of a linear system, including the response to an impulse, also is finite. If the linear system is a filter, then we can completely determine the filter by observing its response to an impulse. A filter determined in this manner is called a *finite-impulse-response* (FIR) *filter*. All the linear filters in this book are FIR filters.

4.3.3 An M-function for Filtering in the Frequency Domain

The filtering steps described in the previous section are used throughout this chapter and parts of the next, so it will be convenient to have available an M-function that accepts as inputs an image and a filter function, handles all the filtering details, and outputs the filtered, cropped image. The following function does that. It is assumed that the filter function has been sized appropriately, as explained in step 4 of the filtering procedure In some applications, it is useful to convert the filtered image to the same class as the input; in others it is necessary to work with a floating point result. The function has the capability to do both.

dftfilt

```
function g = dftfilt(f, H, classout)
%DFTFILT Performs frequency domain filtering.
%   g = DFTFILT(f, H, CLASSOUT) filters f in the frequency domain
%   using the filter transfer function H. The output, g, is the
```

```
%   filtered image, which has the same size as f.
%
%   Valid values of CLASSOUT are
%
%   'original'  The ouput is of the same class as the input.
%               This is the default if CLASSOUT is not included
%               in the call.
%   'fltpoint'  The output is floating point of class single, unless
%               both f and H are of class double, in which case the
%               output also is of class double.
%
%   DFTFILT automatically pads f to be the same size as H. Both f
%   and H must be real. In addition, H must be an uncentered,
%   circularly-symmetric filter function.

% Convert the input to floating point.
[f, revertClass] = tofloat(f);

% Obtain the FFT of the padded input.
F = fft2(f, size(H, 1), size(H, 2));

% Perform filtering.
g = ifft2(H.*F);

% Crop to original size.
g = g(1:size(f, 1), 1:size(f, 2)); % g is of class single here.

% Convert the output to the same class as the input if so specified.
if nargin == 2 || strcmp(classout, 'original')
   g = revertClass(g);
elseif strcmp(classout, 'fltpoint')
   return
else
   error('Undefined class for the output image.')
end
```

Techniques for generating frequency-domain filters are discussed in the following three sections.

4.4 Obtaining Frequency Domain Filters from Spatial Filters

In general, filtering in the spatial domain is more efficient computationally than frequency domain filtering when the filters are small. The definition of *small* is a complex question whose answer depends on such factors as the machine and algorithms used, and on issues such as the size of buffers, how well complex data are handled, and a host of other factors beyond the scope of this discussion. A comparison by Brigham [1988] using 1-D functions shows that filtering using an FFT algorithm can be faster than a spatial implementation when the filters have on the order of 32 or more elements, so the numbers in question

are not large. Thus, it is useful to know how to convert a spatial filter into an equivalent frequency domain filter in order to obtain meaningful comparisons between the two approaches.

We are interested in this section on two major topics: (1) how to convert spatial filters into equivalent frequency domain filters; and (2) how to compare the results between spatial domain filtering using function `imfilter`, and frequency domain filtering using the techniques discussed in the previous section. Because, as explained in detail in Section 3.4.1, `imfilter` uses correlation and the origin of the filter is considered at its center, some preprocessing is required to make the two approaches equivalent. Image Processing Toolbox function `freqz2` does this, and outputs the corresponding filter in the frequency domain.

Function `freqz2` computes the frequency response of FIR filters which, as mentioned at the end of Section 4.3.2, are the only linear filters considered in this book. The result is the desired filter in the frequency domain. The syntax relevant in the present discussion is

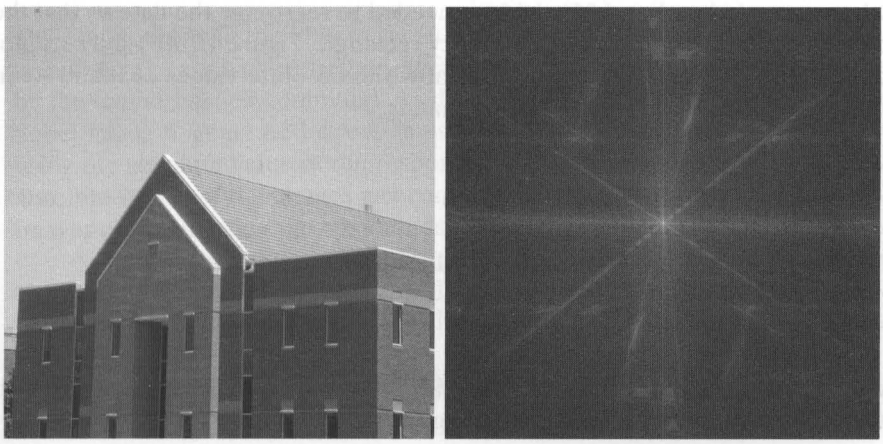
```
H = freqz2(h, R, C)
```



where `h` is a 2-D spatial filter and `H` is the corresponding 2-D frequency domain filter. Here, `R` is the number of rows, and `C` the number of columns that we wish filter `H` to have. Generally, we let `R = PQ(1)` and `C = PQ(2)`, as explained in Section 4.3.1. If `freqz2` is written without an output argument, the absolute value of `H` is displayed on the MATLAB desktop as a 3-D perspective plot. The mechanics involved in using function `freqz2` are best explained by an example.

EXAMPLE 4.2: A comparison of filtering in the spatial and frequency domains.

■ Consider the 600 × 600-pixel image, `f`, in Fig. 4.9(a). In what follows, we generate the frequency domain filter, `H`, corresponding to the Sobel spatial filter that enhances vertical edges (Table 3.5). Then, using `imfilter`, we compare the result of filtering `f` in the spatial domain with the Sobel mask against the result obtained by performing the equivalent process in the frequency



a b

FIGURE 4.9 (a) A gray-scale image. (b) Its Fourier spectrum.

domain. In practice, filtering with a small filter like a Sobel mask would be implemented directly in the spatial domain, as mentioned earlier. However, we selected this filter for demonstration purposes because its coefficients are simple and because the results of filtering are intuitive and straightforward to compare. Larger spatial filters are handled in the same manner.

Figure 4.9(b) is the Fourier spectrum of `f`, obtained in the usual manner:

```
>> f = tofloat(f);
>> F = fft2(f);
>> S = fftshift(log(1 + abs(F)));
>> imshow(S, [ ])
```

Next, we generate the spatial filter using function `fspecial`:

```
h = fspecial('sobel')'
h =
    1    0   -1
    2    0   -2
    1    0   -1
```

To view a plot of the corresponding frequency domain filter we type

```
>> freqz2(h)
```

Figure 4.10(a) shows the result, with the axes suppressed (techniques for obtaining perspective plots are discussed in Section 4.5.3). The filter itself was obtained using the commands:

```
>> PQ = paddedsize(size(f));
>> H = freqz2(h, PQ(1), PQ(2));
>> H1 = ifftshift(H);
```

where, as noted earlier, `ifftshift` is needed to rearrange the data so that the origin is at the top, left of the frequency rectangle. Figure 4.10(b) shows a plot of `abs(H1)`. Figures 4.10(c) and (d) show the absolute values of `H` and `H1` in image form, displayed using the commands

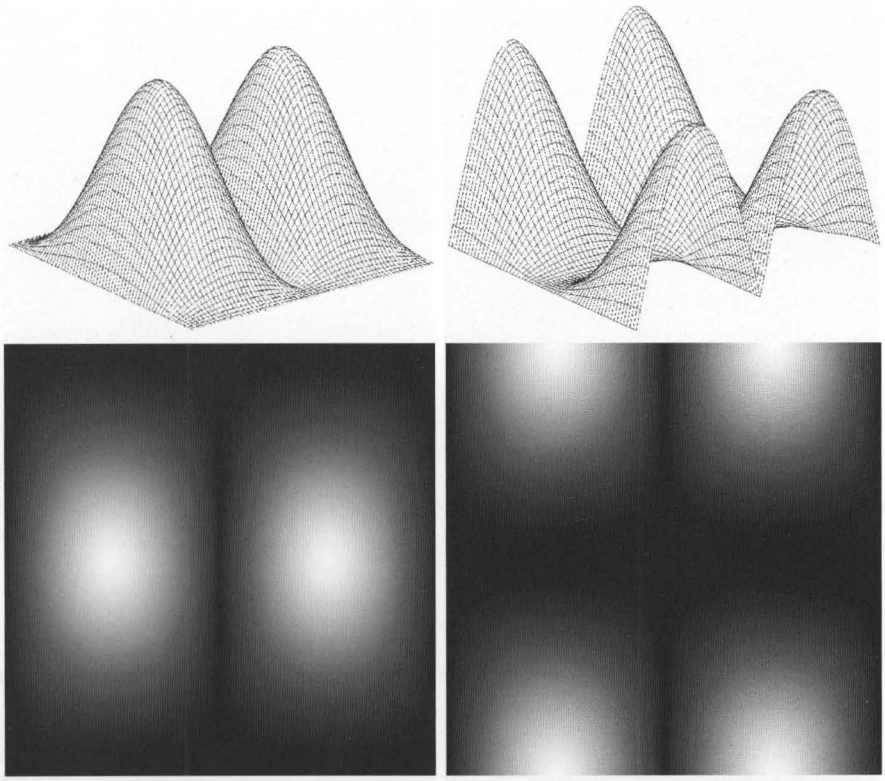
```
>> imshow(abs(H), [ ])
>> figure, imshow(abs(H1), [ ])
```

Because `f` is floating point, `imfilter` will produce a floating point result, as explained in Section 3.4.1. Floating point is required for some of the following operations.

Next, we generate the filtered images. In the spatial domain we use

```
>> gs = imfilter(f, h);
```

which pads the border of the image with 0s by default. The filtered image obtained by frequency domain processing is given by



a b
c d

FIGURE 4.10
(a) Absolute value of the frequency domain filter corresponding to a vertical Sobel spatial filter. (b) The same filter after processing with function `ifftshift`. Figures (c) and (d) show the filters as images.

```
>> gf = dftfilt(f, H1);
```

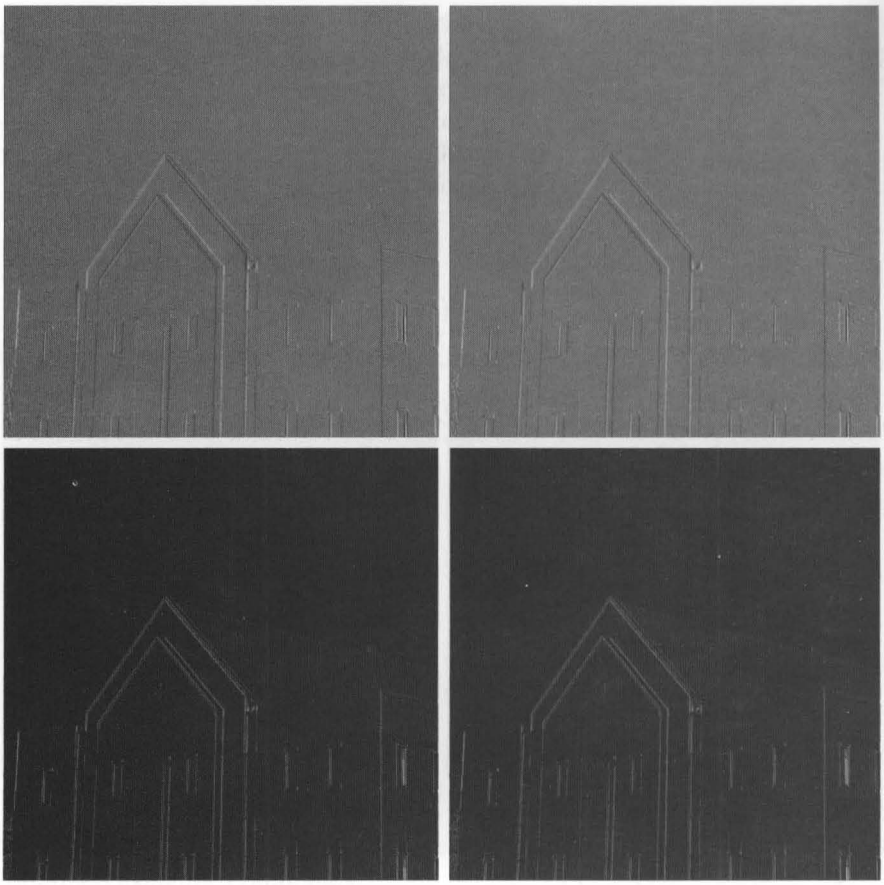
Figures 4.11(a) and (b) show the result of the commands:

```
>> imshow(gs, [ ])
>> figure, imshow(gf, [ ])
```

The gray tonality in the images is caused by the fact that both `gs` and `gf` have negative values, which causes the average value of the images to be increased by the scaled `imshow` command. As discussed in Sections 7.6.1 and 11.1.3, the Sobel mask, `h`, generated above is used to detect vertical edges in an image using the absolute value of the response. Thus, it is more relevant to show the absolute values of the images just computed. Figures 4.11(c) and (d) show the images obtained using the commands

```
>> figure, imshow(abs(gs), [ ])
>> figure, imshow(abs(gf), [ ])
```

The edges can be seen more clearly by creating a thresholded binary image:



a b
c d

FIGURE 4.11
(a) Result of filtering Fig. 4.9(a) in the spatial domain with a vertical Sobel mask. (b) Result obtained in the frequency domain using the filter shown in Fig. 4.10(b). Figures (c) and (d) are the absolute values of (a) and (b), respectively.

```
>> figure, imshow(abs(gs) > 0.2*abs(max(gs(:))))
>> figure, imshow(abs(gf) > 0.2*abs(max(gf(:))))
```

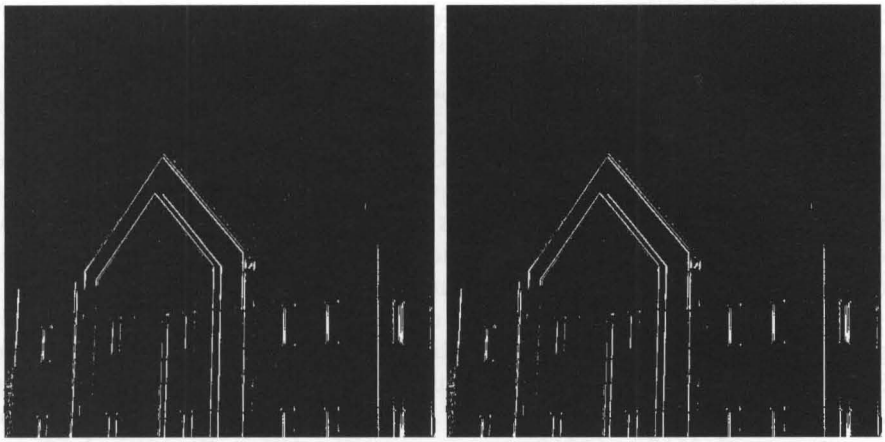
where the 0.2 multiplier was selected to show only the edges with strength greater than 20% of the maximum values of `gs` and `gf`. Figures 4.12(a) and (b) show the results.

The images obtained using spatial and frequency domain filtering are for all practical purposes identical, a fact that we confirm by computing their difference:

```
>> d = abs(gs - gf);
```

The maximum difference is

```
>> max(d(:))
```



a b

FIGURE 4.12 Thresholded versions of Figs. 4.11(c) and (d), respectively, to show the principal edges more clearly.

```
ans =
    1.2973e-006
```

which is negligible in the context of the present application. The minimum difference is

```
>> min(d(:))
ans =
    0
```

The approach just explained can be used to implement in the frequency domain the spatial filtering approach discussed in Sections 3.4.1 and 3.5.1, as well as any other FIR spatial filter of arbitrary size. ■

4.5 Generating Filters Directly in the Frequency Domain

In this section, we illustrate how to implement filter functions directly in the frequency domain. We focus on circularly symmetric filters that are specified as various functions of the distance from the center of the filters. The custom M-functions developed to implement these filters are a foundation that is easily extendable to other functions within the same framework. We begin by implementing several well-known smoothing (lowpass) filters. Then, we show how to use several of MATLAB's wireframe and surface plotting capabilities for filter visualization. After that we discuss sharpening (highpass) filters, and conclude the chapter with a development of selective filtering techniques.

4.5.1 Creating Meshgrid Arrays for Use in Implementing Filters in the Frequency Domain

Central to the M-functions in the following discussion is the need to compute distance functions from any point to a specified point in the frequency rectangle. Because FFT computations in MATLAB assume that the origin of the transform is at the top, left of the frequency rectangle, our distance computations are with respect to that point. As before, the data can be rearranged for visualization purposes (so that the value at the origin is translated to the center of the frequency rectangle) by using function `fftshift`.

The following M-function, which we call `dftuv`, provides the necessary meshgrid arrays for use in distance computations and other similar applications. (See Section 2.10.5 for an explanation of function `meshgrid` used in the following code.). The meshgrid arrays generated by `dftuv` are in the order required for processing with `fft2` or `ifft2`, so rearranging the data is not required.

`dftuv`

```
function [U, V] = dftuv(M, N)
%DFTUV Computes meshgrid frequency matrices.
%   [U, V] = DFTUV(M, N) computes meshgrid frequency matrices U and
%   V.  U and V are useful for computing frequency-domain filter
%   functions that can be used with DFTFILT.  U and V are both
%   M-by-N and of class single.

% Set up range of variables.
u = single(0:(M - 1));
v = single(0:(N - 1));

% Compute the indices for use in meshgrid.
idx = find(u > M/2);
u(idx) = u(idx) - M;
idy = find(v > N/2);
v(idy) = v(idy) - N;

% Compute the meshgrid arrays.
[V, U] = meshgrid(v, u);
```

Function `find` is discussed in Section 5.2.2.

EXAMPLE 4.3: Using function `dftuv`.

■ As an illustration, the following commands compute the distance squared from every point in a rectangle of size 8×5 to the origin of the rectangle:

```
>> [U, V] = dftuv(8, 5);
>> DSQ = U.^2 + V.^2
DSQ =
     0     1     4     4     1
     1     2     5     5     2
     4     5     8     8     5
     9    10    13    13    10
    16    17    20    20    17
     9    10    13    13    10
     4     5     8     8     5
     1     2     5     5     2
```

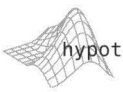
Note that the distance is 0 at the top, left, and the larger distances are in the center of the frequency rectangle, following the basic format explained in Fig. 4.2(a). We can use function `fftshift` to obtain the distances with respect to the center of the frequency rectangle,

```
>> fftshift(DSQ)
ans =
    20    17    16    17    20
    13    10     9    10    13
     8     5     4     5     8
     5     2     1     2     5
     4     1     0     1     4
     5     2     1     2     5
     8     5     4     5     8
    13    10     9    10    13
```

The distance is now 0 at coordinates (5, 3), and the array is symmetric about this point.

While on the subject of distances, we mention that function `hypot` performs the same computation as `D = sqrt(U.^2 + V.^2)`, but faster. For example, letting `U = V = 1024` and using function `timeit` (see Section 2.10.5), we find that `hypot` computes `D` nearly 100 times faster than the "standard" way. The syntax for `hypot` is:

```
D = hypot(U, V)
```



We use `hypot` extensively in the following sections. ■

4.5.2 Lowpass (Smoothing) Frequency Domain Filters

An *ideal lowpass filter* (ILPF) has the transfer function

$$H(u,v) = \begin{cases} 1 & \text{if } D(u,v) \le D_0 \\ 0 & \text{if } D(u,v) > D_0 \end{cases}$$

where D_0 is a positive number and $D(u,v)$ is the distance from point (u,v) to the center of the filter. The locus of points for which $D(u,v) = D_0$ is a circle. Because filter $H(u,v)$ multiplies the Fourier transform of an image, we see that an ideal filter "cuts off" (multiplies by 0) all components of $F(u,v)$ outside the circle and leaves unchanged (multiplies by 1) all components on, or inside, the circle. Although this filter is not realizable in analog form using electronic components, it certainly can be simulated in a computer using the preceding transfer function. The properties of ideal filters often are useful in explaining phenomena such as ringing and wraparound error.

A *Butterworth lowpass filter* (BLPF) of order n, with a cutoff frequency at a distance D_0 from the center of the filter, has the transfer function

$$H(u,v) = \frac{1}{1 + \left[D(u,v)/D_0\right]^{2n}}$$

Unlike the ILPF, the BLPF transfer function does not have a sharp discontinuity at D_0. For filters with smooth transfer functions, it is customary to define a cutoff frequency locus at points for which $H(u,v)$ is down to a specified fraction of its maximum value. In the preceding equation, $H(u,v) = 0.5$ (down 50% from its maximum value of 1) when $D(u,v) = D_0$.

The transfer function of a *Gaussian lowpass filter* (GLPF) is given by

$$H(u,v) = e^{-D^2(u,v)/2\sigma^2}$$

where σ is the standard deviation. By letting $\sigma = D_0$ we obtain the following expression in terms of the cutoff parameter

$$H(u,v) = e^{-D^2(u,v)/2D_0^2}$$

When $D(u,v) = D_0$ the filter is down to 0.607 of its maximum value of 1. The preceding filters are summarized in Table 4.1.

EXAMPLE 4.4: Lowpass filtering.

■ As an illustration, we apply a Gaussian lowpass filter to the 500 × 500-pixel image, `f`, in Fig. 4.13(a). We use a value of D_0 equal to 5% of the padded image width. With reference to the filtering steps discussed in Section 4.3.2, we write

```
>> [f, revertclass] = tofloat(f);
>> PQ = paddedsize(size(f));
>> [U, V] = dftuv(PQ(1), PQ(2));
>> D = hypot(U, V);
>> D0 = 0.05*PQ(2);
>> F = fft2(f, PQ(1), PQ(2)); % Needed for the spectrum.
>> H = exp(-(D.^2)/(2*(D0^2)));
>> g = dftfilt(f, H);
>> g = revertclass(g);
```

To view the filter as an image [Fig. 4.13(b)] we center it using `fftshift`:

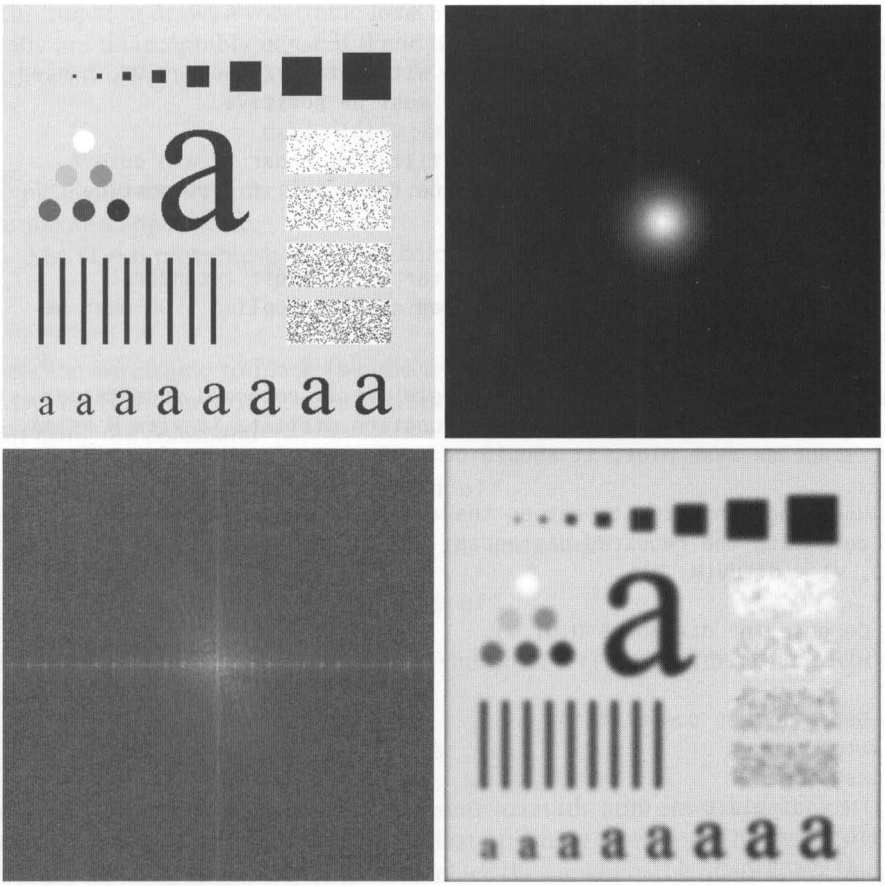
```
>> figure, imshow(fftshift(H))
```

Similarly, the spectrum can be displayed as an image [Fig. 4.13(c)] by typing

```
>> figure, imshow(log(1 + abs(fftshift(F))), [ ])
```

TABLE 4.1 Lowpass filters. D_0 is the cutoff frequency and n is the order of the Butterworth filter.

Ideal	Butterworth	Gaussian
$H(u,v) = \begin{cases} 1 & \text{if } D(u,v) \le D_0 \\ 0 & \text{if } D(u,v) > D_0 \end{cases}$	$H(u,v) = \frac{1}{1 + \left[D(u,v)/D_0\right]^{2n}}$	$H(u,v) = e^{-D^2(u,v)/2D_0^2}$



a b
c d

FIGURE 4.13 Lowpass filtering. (a) Original image. (b) Gaussian lowpass filter shown as an image. (c) Spectrum of (a). (d) Filtered image.

Finally, Fig. 4.13(d) shows the output image, displayed using the command

```
>> figure, imshow(g)
```

As expected, this image is a blurred version of the original. ■

The following function generates the transfer functions of the lowpass filters in Table 4.1.

lpfilter

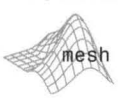
```
function H = lpfilter(type, M, N, D0, n)
%LPFILTER Computes frequency domain lowpass filters.
%   H = LPFILTER(TYPE, M, N, D0, n) creates the transfer function of
%   a lowpass filter, H, of the specified TYPE and size (M-by-N). To
%   view the filter as an image or mesh plot, it should be centered
%   using H = fftshift(H).
%
```

```
%   Valid values for TYPE, DO, and n are:
%
%   'ideal'     Ideal lowpass filter with cutoff frequency DO. n need
%               not be supplied.  DO must be positive.
%
%   'btw'       Butterworth lowpass filter of order n, and cutoff
%               DO.  The default value for n is 1.0.  DO must be
%               positive.
%
%   'gaussian'  Gaussian lowpass filter with cutoff (standard
%               deviation) DO.  n need not be supplied.  DO must be
%               positive.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).

% Use function dftuv to set up the meshgrid arrays needed for
% computing the required distances.
[U, V] = dftuv(M, N);

% Compute the distances D(U, V).
D = hypot(U, V);

% Begin filter computations.
switch type
case 'ideal'
   H = single(D <= DO);
case 'btw'
   if nargin == 4
      n = 1;
   end
   H = 1./(1 + (D./DO).^(2*n));
case 'gaussian'
   H = exp(-(D.^2)./(2*(DO^2)));
otherwise
   error('Unknown filter type.')
end
```

Function `lpfilter` is used again in Section 4.6 as the basis for generating highpass filters.

4.5.3 Wireframe and Surface Plotting

Function `mesh` only supports classes `double` and `uint8`. All our filters are of class `single` to conserve memory so, if `H` is a filter function, we use the syntax `mesh(double(H))`.



Plots of functions of one variable were introduced in Section 3.3.1. In the following discussion we introduce 3-D wireframe and surface plots, which are useful for visualizing 2-D filters. The easiest way to draw a wireframe plot of an $M \times N$, 2-D function, `H`, is to use function `mesh`, which has the basic syntax

```
mesh(H)
```

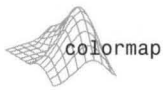
This function draws a wireframe for `x = 1:M` and `y = 1:N`. Wireframe plots typically are unacceptably dense if `M` and `N` are large, in which case we plot every `k`th point using the syntax

```
mesh(H(1:k:end, 1:k:end))
```

Typically, 40 to 60 points along each axis provide a good balance between resolution and appearance.

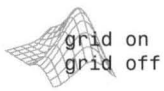
MATLAB plots mesh figures in color by default. The command

```
colormap([0 0 0])
```



sets the wireframe to black (we discuss function `colormap` in Section 7.1.2). MATLAB also superimposes a grid and axes on a mesh plot. The grid is turned off using the command

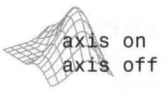
```
grid off
```



`grid off` turns the grid off; `grid on` turns it on.

Similarly, the axes are turned off using the command†

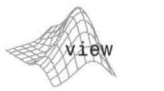
```
axis off
```



`axis on` turns the axis on; `axis off` turns it off.

Finally, the viewing point (location of the observer) is controlled by function `view`, which has the syntax

```
view(az, el)
```



As Fig. 4.14 shows, `az` and `el` represent azimuth and elevation angles (in degrees), respectively. The arrows indicate positive direction. The default values are `az = -37.5` and `el = 30`, which place the viewer in the quadrant defined

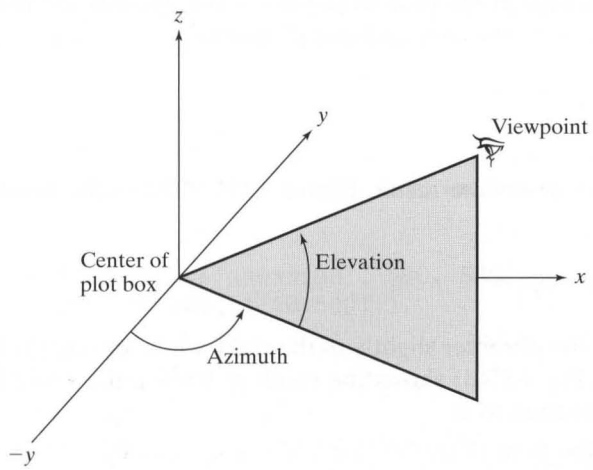


FIGURE 4.14
Viewing geometry for function `view`.

†Turning the axis off (on) turns the grid off (on) also. The reverse is not true.

by the $-x$ and $-y$ axes, and looking into the quadrant defined by the positive x and y axes in Fig. 4.14.

To determine the current viewing geometry, type

```
>> [az, el] = view;
```

To set the viewpoint to the default values, type

```
>> view(3)
```

The viewpoint can be modified interactively by clicking on the **Rotate 3D** button in the figure window's toolbar and then clicking and dragging in the figure window.

As discussed in Section 7.1.1, it is possible to specify the viewer location in Cartesian coordinates, (x, y, z), which is ideal when working with RGB data. However, for general plot-viewing purposes, the method just discussed involves only two parameters and is more intuitive.

EXAMPLE 4.5: Wireframe plotting.

■ Consider a Gaussian lowpass filter similar to the one in Example 4.4:

```
>> H = fftshift(lpfilter('gaussian', 500, 500, 50));
```

Figure 4.15(a) shows the wireframe plot produced by the commands

```
>> mesh(double(H(1:10:500, 1:10:500)))
>> axis tight
```

where the `axis` command is as described in Section 3.3.1.

As noted earlier in this section, the wireframe is in color by default, transitioning from blue at the base to red at the top. We convert the plot lines to black and eliminate the axes and grid by typing

```
>> colormap([0 0 0])
>> axis off
```

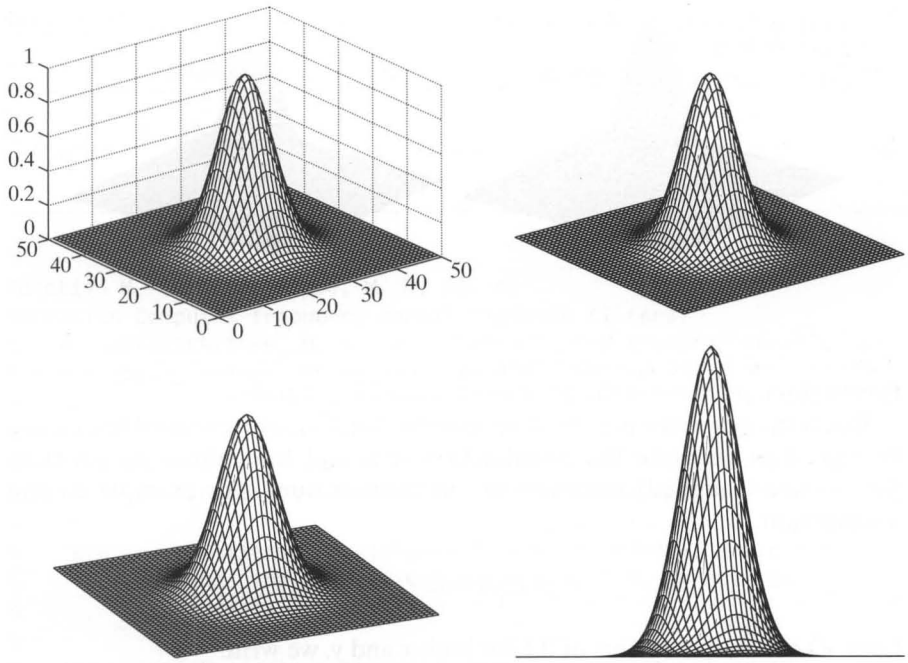
Figure 4.15(b) shows the result. Figure 4.15(c) shows the result of the command

```
>> view(-25, 30)
```

which moved the observer slightly to the right, while leaving the elevation constant. Finally, Fig. 4.15(d) shows the result of leaving the azimuth at `-25` and setting the elevation to `0`:

```
>> view(-25, 0)
```

This example shows the significant plotting power of function `mesh`. ■

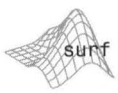


a b
c d

FIGURE 4.15
(a) A plot obtained using function `mesh`. (b) Axes and grid removed. (c) A different perspective view obtained using function `view`. (d) Another view obtained using the same function.

Sometimes it is desirable to plot a function as a surface instead of as a wireframe. Function `surf` does this. Its basic syntax is

```
surf(H)
```



Function `surf` only supports classes `double` and `uint8`. All our filters are of class `single` to conserve memory, so, if `H` is a filter function, we use the syntax `surf(double(H))`.

This function produces a plot identical to `mesh`, with the exception that the quadrilaterals in the mesh are filled with colors (this is called *faceted shading*). To convert the colors to gray, we use the command

```
colormap(gray)
```

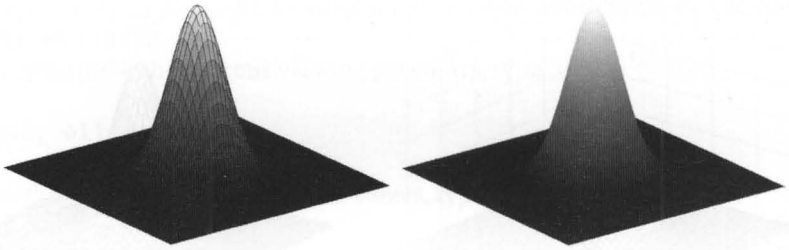
The `axis`, `grid`, and `view` functions work in the same way as described earlier for `mesh`. For example, Fig. 4.16(a) resulted from the following sequence of commands:

```
>> H = fftshift(lpfilter('gaussian', 500, 500, 50));
>> surf(double(H(1:10:500, 1:10:500)))
>> axis tight
>> colormap(gray)
>> axis off
```

The faceted shading can be smoothed and the mesh lines eliminated by interpolation using the command

```
shading interp
```





a b

FIGURE 4.16
(a) Plot obtained using function `surf`. (b) Result of using the command `shading interp`.

Typing this command at the prompt produced Fig. 4.16(b).

When the objective is to plot an analytic function of two variables, we use `meshgrid` to generate the coordinate values and from these we generate the discrete (sampled) matrix to use in `mesh` or `surf`. For example, to plot the function

$$f(x, y) = xe^{-(x^2+y^2)}$$

from -2 to 2 in increments of 0.1 for both x and y, we write

```
>> [Y, X] = meshgrid(-2:0.1:2, -2:0.1:2);
>> Z = X.*exp(-X.^2 - Y.^2);
```

and then use `mesh(Z)` or `surf(Z)` as before. Recall from the discussion in Section 2.10.5 that columns `(Y)` are listed first and rows `(X)` second in function `meshgrid`.

4.6 Highpass (Sharpening) Frequency Domain Filters

Just as lowpass filtering blurs an image, the opposite process, *highpass filtering*, sharpens the image by attenuating the low frequencies and leaving the high frequencies of the Fourier transform relatively unchanged. In this section we consider several approaches to highpass filtering.

Given the transfer function $H_{\text{LP}}(u, v)$ of a lowpass filter, the transfer function of the corresponding highpass filter is given by

$$H_{\text{HP}}(u, v) = 1 - H_{\text{LP}}(u, v)$$

Table 4.2 shows the highpass filter transfer functions corresponding to the lowpass filters in Table 4.1.

4.6.1 A Function for Highpass Filtering

Based on the preceding equation, we can use function `lpfilter` from the previous section to construct a function that generates highpass filters, as follows:

TABLE 4.2 Highpass filters. D_0 is the cutoff frequency and n is the order of the Butterworth filter.

Ideal	Butterworth	Gaussian
$H(u,v) = \begin{cases} 0 & \text{if } D(u,v) \le D_0 \\ 1 & \text{if } D(u,v) > D_0 \end{cases}$	$H(u,v) = \dfrac{1}{1 + \left[D_0/D(u,v)\right]^{2n}}$	$H(u,v) = 1 - e^{-D^2(u,v)/2D_0^2}$

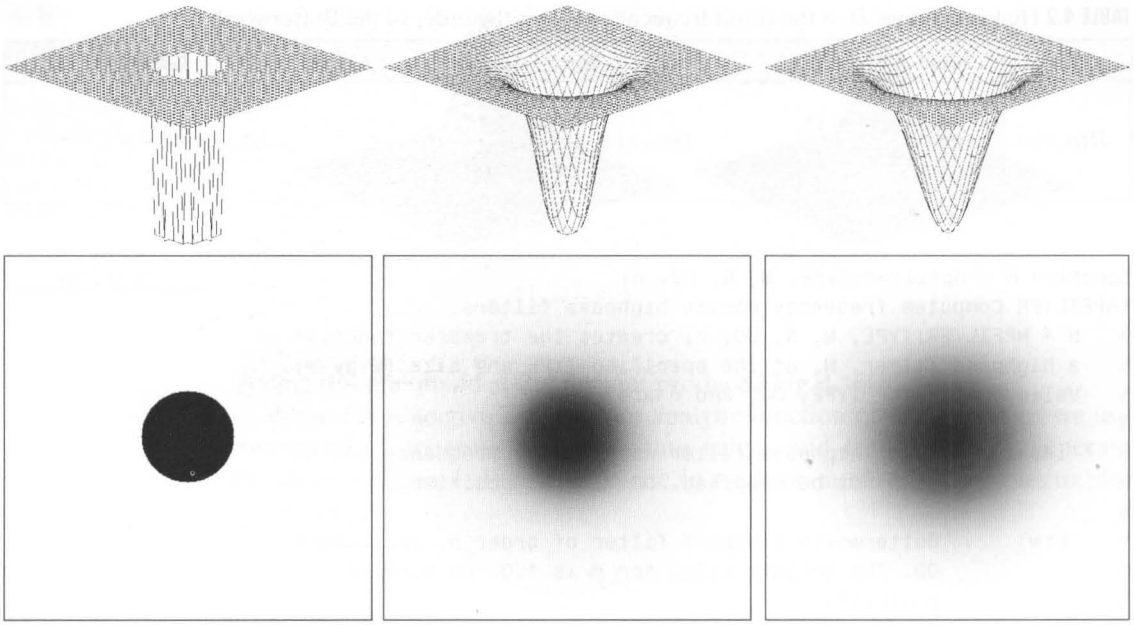
hpfilter

```
function H = hpfilter(type, M, N, D0, n)
%HPFILTER Computes frequency domain highpass filters.
%   H = HPFILTER(TYPE, M, N, D0, n) creates the transfer function of
%   a highpass filter, H, of the specified TYPE and size (M-by-N).
%   Valid values for TYPE, D0, and n are:
%
%   'ideal'     Ideal highpass filter with cutoff frequency D0. n
%               need not be supplied. D0 must be positive.
%
%   'btw'       Butterworth highpass filter of order n, and cutoff
%               D0. The default value for n is 1.0. D0 must be
%               positive.
%
%   'gaussian' Gaussian highpass filter with cutoff (standard
%               deviation) D0. n need not be supplied. D0 must be
%               positive.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).

% The transfer function Hhp of a highpass filter is 1 - Hlp,
% where Hlp is the transfer function of the corresponding lowpass
% filter. Thus, we can use function lpfilter to generate highpass
% filters.

if nargin == 4
   n = 1; % Default value of n.
end

% Generate highpass filter.
Hlp = lpfilter(type, M, N, D0, n);
H = 1 - Hlp;
```

EXAMPLE 4.6: Highpass filters.

■ Figure 4.17 shows plots and images of ideal, Butterworth, and Gaussian highpass filters. The plot in Fig. 4.17(a) was generated using the commands

```
>> H = fftshift(hpfilter('ideal', 500, 500, 50));
>> mesh(double(H(1:10:500, 1:10:500)));
>> axis tight
>> colormap([0 0 0])
```



a b c
d e f

FIGURE 4.17 Top row: Perspective plots of ideal, Butterworth, and Gaussian highpass filters. Bottom row: Corresponding images. White represents 1 and black is 0.

```
>> axis off
```

The corresponding image in Fig. 4.17(d) was generated using the command

```
>> figure, imshow(H, [ ])
```

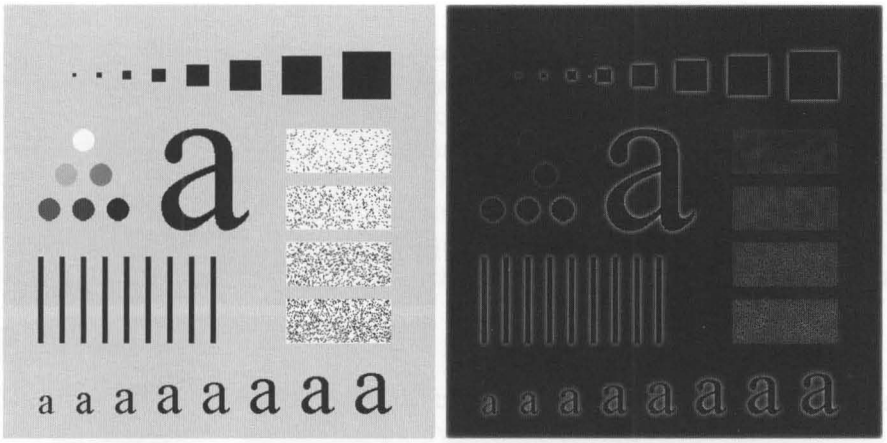
Similar commands using the same value for D_0 yielded the rest of Fig. 4.17 (the Butterworth filter is of order 2). ■

EXAMPLE 4.7: Highpass filtering.

■ Figure 4.18(a) is the same test pattern, `f`, from Fig. 4.13(a). Figure 4.18(b), obtained using the following commands, shows the result of applying a Gaussian highpass filter to `f` in the frequency domain:

```
>> PQ = paddedsize(size(f));
>> D0 = 0.05*PQ(1);
>> H = hpfilter('gaussian', PQ(1), PQ(2), D0);
>> g = dftfilt(f, H);
>> figure, imshow(g)
```

As Fig. 4.18(b) shows, edges and other sharp intensity transitions in the image were enhanced. However, because the average value of an image is given by



a b

FIGURE 4.18
(a) Original image. (b) Result of Gaussian highpass filtering.

$F(0,0)$, and the highpass filters discussed thus far zero-out the origin of the Fourier transform, the image has lost most of the gray tonality present in the original. This problem is addressed in the following section. ■

4.6.2 High-Frequency Emphasis Filtering

As mentioned in Example 4.7, highpass filters zero out the dc term, thus reducing the average value of an image to 0. An approach used to compensate for this is to add an offset to a highpass filter. When an offset is combined with multiplying the filter by a constant greater than 1, the approach is called *high-frequency emphasis* filtering because the constant multiplier highlights the high frequencies. The multiplier increases the amplitude of the low frequencies also, but the low-frequency effects on enhancement are less than those due to high frequencies, provided that the offset is small compared to the multiplier. High-frequency emphasis filters have the transfer function

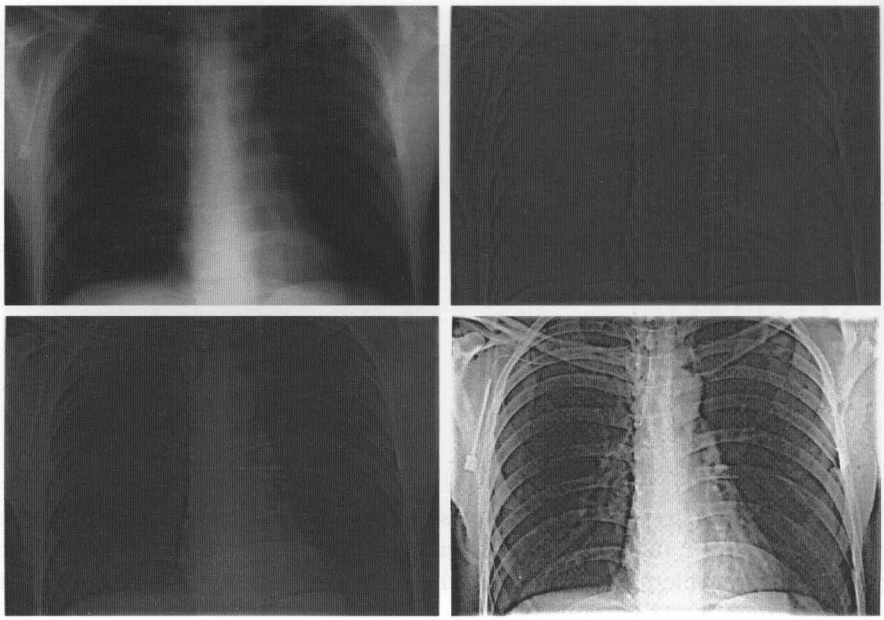
$$H_{\text{HFE}}(u,v) = a + bH_{\text{HP}}(u,v)$$

where a is the offset, b is the multiplier, and $H_{\text{HP}}(u,v)$ is the transfer function of a highpass filter.

EXAMPLE 4.8: Combining high-frequency emphasis and histogram equalization.

■ Figure 4.19(a) is a digital chest X-ray image. X-ray imagers cannot be focused in the same manner as optical lenses, so the resulting images generally tend to be slightly blurred. The objective of this example is to sharpen Fig. 4.19(a). Because the intensity levels in this particular image are biased toward the dark end of the gray scale, we also take this opportunity to give an example of how spatial domain processing can be used to complement frequency domain filtering.

Figure 4.19(b) shows the result of filtering Fig. 4.19(a) with a Butterworth highpass filter of order 2, and a value of D_0 equal to 5% of the vertical dimension of the padded image. Highpass filtering is not overly sensitive to



a b
c d

FIGURE 4.19
High-frequency emphasis filtering. (a) Original image. (b) Highpass filtering result. (c) High-frequency emphasis result. (d) Image (c) after histogram equalization. (Original image courtesy of Dr. Thomas R. Gest, Division of Anatomical Sciences, University of Michigan Medical School.)

the value of D_0, provided that the radius of the filter is not so small that frequencies near the origin of the transform are passed. As expected, the filtered result is rather featureless, but it shows faintly the principal edges in the image. The only way a nonzero image can have a zero average value is if some of its intensity values are negative. This is the case in the filtered result in Fig. 4.19(b). For this reason, we had to use the `fltpoint` option in function `dftfilt` to obtain a floating point result. If we had not, the negative values would have been clipped in the default conversion to `uint8` (the class of the input image), thus losing some of the faint detail. Using function `gscale` takes into account negative values, thus preserving these details.

The advantage of high-emphasis filtering (with $a = 0.5$ and $b = 2.0$ in this case) is shown in Fig. 4.19(c), in which the gray-level tonality due to the low-frequency components was retained. The following commands were used to generate the processed images in Fig. 4.19, where `f` denotes the input image [the last command generated Fig. 4.19(d)]:

```
>> PQ = paddedsize(size(f));
>> D0 = 0.05*PQ(1);
>> HBW = hpfilter('btw', PQ(1), PQ(2), D0, 2);
>> H = 0.5 + 2*HBW;
>> gbw = dftfilt(f, HBW, 'fltpoint');
>> gbw = gscale(gbw);
>> ghf = dftfilt(f, H, 'fltpoint');
>> ghf = gscale(ghf);
>> ghe = histeq(ghf, 256);
```

As indicated in Section 3.3.2, an image characterized by intensity levels in a narrow range of the gray scale is a candidate for histogram equalization. As Fig. 4.19(d) shows, this indeed was an appropriate method to further enhance the image in this example. Note the clarity of the bone structure and other details that simply are not visible in any of the other three images. The final enhanced image appears a little noisy, but this is typical of X-ray images when their gray scale is expanded. The result obtained using a combination of high-frequency emphasis and histogram equalization is superior to the result that would be obtained by using either method alone. ■

4.7 Selective Filtering

The filters introduced in the previous two sections operate over the entire frequency rectangle. As you will see shortly, there are applications that require that bands or small regions in the frequency rectangle be filtered. Filters in the first category are called *bandreject* or *bandpass filters,* depending on their function. Similarly, filters in the second category are called *notchreject* or *notchpass filters*.

4.7.1 Bandreject and Bandpass Filters

These filters are easy to construct using lowpass and highpass filter forms. As with those filters, we obtain a bandpass filter $H_{BP}(u, v)$ from a given a bandreject filter $H_{BR}(u, v)$, using the expression

$$H_{BP}(u, v) = 1 - H_{BR}(u, v)$$

Table 4.3 shows expressions for ideal, Butterworth, and Gaussian bandreject filters. Parameter W is the true width of the band only for the ideal filter. For the Gaussian filter the transition is smooth, with W acting roughly as a cutoff frequency. For the Butterworth filter the transition is smooth also, but W and n act together to determine the broadness of the band, which increases as a function of increasing W and n. Figure. 4.20 shows images of a bandreject Gaussian filter and its corresponding bandpass filter obtained using the following function, which implements both bandreject and bandpass filters.

bandfilter

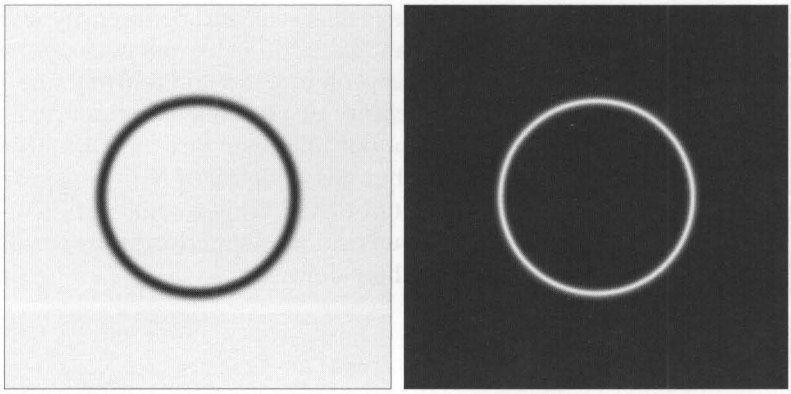
```
function H = bandfilter(type, band, M, N, D0, W, n)
%BANDFILTER Computes frequency domain band filters.
%
```

TABLE 4.3 Bandreject filters. W is the "width" of the band, $D(u, v)$ is the distance from the center of the filter, D_0 is the radius of the center of the band, and n is the order of the Butterworth filter.

Ideal	Butterworth	Gaussian
$H(u,v) = \begin{cases} 0 & \text{for } D_0 - \frac{W}{2} \le D(u,v) \le D_0 + \frac{W}{2} \\ 1 & \text{otherwise} \end{cases}$	$H(u,v) = \frac{1}{1 + \left[\frac{WD(u,v)}{D^2(u,v) - D_0^2}\right]^{2n}}$	$H(u,v) = 1 - e^{-\left[\frac{D^2(u,v) - D_0^2}{WD(u,v)}\right]^2}$

a b

FIGURE 4.20
(a) A Gaussian bandreject filter. (b) Corresponding bandpass filter. The filters were generated using $M = N = 800$, $D_0 = 200$, and $W = 20$ in function `bandfilter`.



```
%   Parameters used in the filter definitions (see Table 4.3 in
%   DIPUM 2e for more details about these parameters):
%       M:  Number of rows in the filter.
%       N:  Number of columns in the filter.
%       D0: Radius of the center of the band.
%       W:  "Width" of the band. W is the true width only for
%           ideal filters. For the other two filters this parameter
%           acts more like a smooth cutoff.
%       n:  Order of the Butterworth filter if one is specified. W
%           and n interplay to determine the effective broadness of
%           the reject or pass band. Higher values of both these
%           parameters result in broader bands.
%   Valid values of BAND are:
%
%       'reject'    Bandreject filter.
%
%       'pass'      Bandpass filter.
%
%   One of these two values must be specified for BAND.
%
%   H = BANDFILTER('ideal', BAND, M, N, D0, W) computes an M-by-N
%   ideal bandpass or bandreject filter, depending on the value of
%   BAND.
%
%   H = BANDFILTER('btw', BAND, M, N, D0, W, n) computes an M-by-N
%   Butterworth filter of order n. The filter is either bandpass or
%   bandreject, depending on the value of BAND. The default value of
%   n is 1.
%
%   H = BANDFILTER('gaussian', BAND, M, N, D0, W) computes an M-by-N
%   gaussian filter. The filter is either bandpass or bandreject,
%   depending on BAND.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
```

```
%  image or mesh plot, it should be centered using Hc = fftshift(H).

% Use function dftuv to set up the meshgrid arrays needed for
% computing the required distances.
[U, V] = dftuv(M, N);
% Compute the distances D(U, V).
D = hypot(U, V);
% Determine if need to use default n.
if nargin < 7
   n = 1; % Default BTW filter order.
end

% Begin filter computations. All filters are computed as bandreject
% filters. At the end, they are converted to bandpass if so
% specified. Use lower(type) to protect against the input being
% capitalized.
switch lower(type)
case 'ideal'
   H = idealReject(D, D0, W);
case 'btw'
   H = btwReject(D, D0, W, n);
case 'gaussian'
   H = gaussReject(D, D0, W);
otherwise
   error('Unknown filter type.')
end

% Generate a bandpass filter if one was specified.
if strcmp(band, 'pass')
   H = 1 - H;
end

%-----------------------------------------------------------------------%
function H = idealReject(D, D0, W)
RI = D <= D0 - (W/2);   % Points of region inside the inner
                        % boundary of the reject band are labeled 1.
                        % All other points are labeled 0.

RO = D >= D0 + (W/2);   % Points of region outside the outer
                        % boundary of the reject band are labeled 1.
                        % All other points are labeled 0.

H = tofloat(RO | RI);   % Ideal bandreject filter.

%-----------------------------------------------------------------------%
function H = btwReject(D, D0, W, n)
H = 1./(1 + (((D*W)./(D.^2 - D0^2)).^2*n));

%-----------------------------------------------------------------------%
function H = gaussReject(D, D0, W)
H = 1 - exp(-((D.^2 - D0^2)./(D.*W + eps)).^2);
```



Functions `lower` and `upper` convert their string inputs to lower- and upper-case, respectively.

4.7.2 Notchreject and Notchpass Filters

Notch filters are the most useful of the selective filters. A notch filter rejects (or passes) frequencies in specified neighborhoods about the center of the frequency rectangle. Zero-phase-shift filters must be symmetric about the center, so, for example, a notch with center at a frequency (u_0, v_0) must have a corresponding notch at $(-u_0, -v_0)$. Notchreject filters are formed as products of highpass filters whose centers have been translated to the centers of the notches. The general form involving Q notch pairs is

$$H_{NR}(u,v) = \prod_{k=1}^{Q} H_k(u,v) H_{-k}(u,v)$$

where $H_k(u,v)$ and $H_{-k}(u,v)$ are highpass filters with centers at (u_k, v_k) and $(-u_k, -v_k)$, respectively. These translated centers are specified with respect to the center of the frequency rectangle, $(M/2, N/2)$. Therefore, the distance computations for the filters are given by the expressions

$$D_k(u,v) = \left[(u - M/2 - u_k)^2 + (v - N/2 - v_k)^2\right]^{\frac{1}{2}}$$

and

$$D_{-k}(u,v) = \left[(u - M/2 + u_k)^2 + (v - N/2 + v_k)^2\right]^{\frac{1}{2}}$$

As an example, the following is a Butterworth notchreject filter of order n, consisting of three notch pairs:

$$H_{NR}(u,v) = \prod_{k=1}^{3} \left[\frac{1}{1 + [D_{0k}/D_k(u,v)]^{2n}}\right]\left[\frac{1}{1 + [D_{0k}/D_{-k}(u,v)]^{2n}}\right]$$

The constant D_{0k} is the same for a notch pair, but it can be different for different pairs.

As with bandpass filters, we obtain a notchpass filter from a notchreject filter using the equation

$$H_{NP}(u,v) = 1 - H_{NR}(u,v)$$

The following function, `cnotch`, computes circularly symmetric ideal, Butterworth, and Gaussian notchreject and notchpass filters. Later in this section we discuss rectangular notch filters. Because it is similar to function `bandfilter` in Section 4.7.1, we show only the help section for function `cnotch`. See Appendix C for a complete listing.

```
>> help cnotch
```

cnotch

```
%CNOTCH Generates circularly symmetric notch filters.
%   H = CNOTCH(TYPE, NOTCH, M, N, C, D0, n) generates a notch filter
%   of size M-by-N. C is a K-by-2 matrix with K pairs of frequency
%   domain coordinates (u, v) that define the centers of the filter
```

```
%   notches (when specifying filter locations, remember that
%   coordinates in MATLAB run from 1 to M and 1 to N). Coordinates
%   (u, v) are specified for one notch only. The corresponding
%   symmetric notches are generated automatically. D0 is the radius
%   (cut-off frequency) of the notches. It can be specified as a
%   scalar, in which case it is used in all K notch pairs, or it can
%   be a vector of length K, containing an individual cutoff value
%   for each notch pair. n is the order of the Butterworth filter if
%   one is specified.
%
%       Valid values of TYPE are:
%
%           'ideal'     Ideal notchpass filter. n is not used.
%
%           'btw'       Butterworth notchpass filter of order n. The
%                       default value of n is 1.
%
%           'gaussian'  Gaussian notchpass filter. n is not used.
%
%       Valid values of NOTCH are:
%
%           'reject'    Notchreject filter.
%
%           'pass'      Notchpass filter.
%
%       One of these two values must be specified for NOTCH.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).
```

Function `cnotch` uses custom function `iseven`, which has the syntax

```
E = iseven(A)
```

iseven

where `E` is a logical array the same size as `A`, with 1s (`true`) in the locations corresponding to even numbers in `A` and 0s (`false`) elsewhere. A companion function,

```
O = isodd(A)
```

isodd

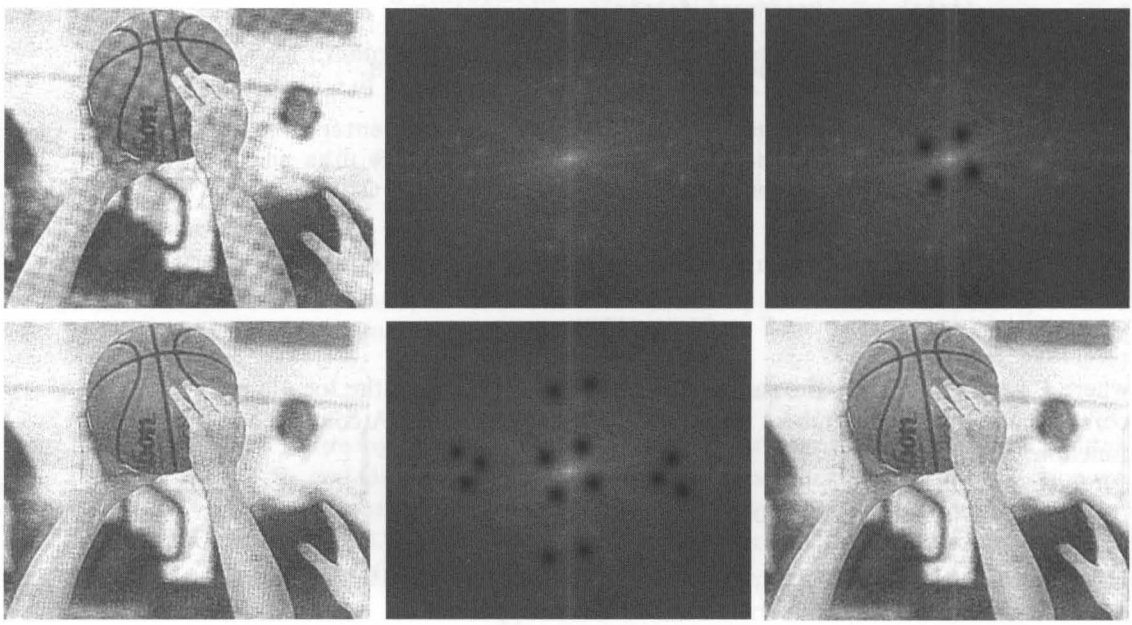
returns 1s in the locations corresponding to odd numbers in `A` and 0s elsewhere. The listings for functions `iseven` and `isodd` are in Appendix C.

EXAMPLE 4.9: Using notch filters to reduce moiré patterns.

■ Newspaper images typically are printed using a spatial resolution of 75 dpi. When such images are scanned at similar resolutions, the results almost invariably exhibit strong moiré patterns. Figure 4.21(a) shows a newspaper image scanned at 72 dpi using a flatbed scanner. A moiré pattern is seen as prominent

periodic interference. The periodic interference leads to strong, localized bursts of energy in the frequency domain, as Fig. 4.21(b) shows. Because the interference is of relatively low frequency, we begin by filtering out the spikes nearest the origin. We do this using function `cnotch`, as follows, where `f` is the scanned image (we used function `imtool` from Section 2.3 to obtain interactively the coordinates of the centers of the energy bursts):

```
>> [M N] = size(f);
>> [f, revertclass] = tofloat(f);
>> F = fft2(f);
>> S = gscale(log(1 + abs(fftshift(F)))); % Spectrum
>> imshow(S)
>> % Use function imtool to obtain the coordinates of the
>> % spikes interactively.
>> C1 = [99 154; 128 163];
>> % Notch filter:
>> H1 = cnotch('gaussian','reject', M, N, C1, 5);
>> % Compute spectrum of the filtered transform and show it as
>> % an image.
>> P1 = gscale(fftshift(H1).*(tofloat(S)));
>> figure, imshow(P1)
```



a b c
d e f

FIGURE 4.21 (a) Scanned, 72 dpi newspaper image of size 232 × 288 pixels corrupted by a moiré pattern. (b) Spectrum. (c) Gaussian notch filters applied to the low-frequency bursts caused by the moiré pattern. (d) Filtered result. (e) Using more filters to eliminate higher frequency "structured" noise. (f) Filtered result.

```
>> % Filter image.
>> g1 = dftfilt(f, H1);
>> g1 = revertclass(g1);
>> figure, imshow(g1)
```

Figure 4.21(c) shows the spectrum with the notch filters superimposed on it. The cutoff values were selected just large enough to encompass the energy bursts, while removing as little as possible from the transform. Figure 4.21(d) shows image g, the filtered result. As you can see, notch filtering reduced the prominence of the moiré pattern to an imperceptible level.

Careful analysis of, for example, the shooter's forearms in Fig. 4.21(d), reveals a faint high-frequency interference associated with the other high-energy bursts in Fig. 4.21(b). The following additional notch filtering operations are an attempt to reduce the contribution of those bursts:

```
>> % Repeat with the following C2 to reduce the higher
>> % frequency interference components.
>> C2 = [99 154; 128 163; 49 160; 133 233; 55 132; 108 225; 112 74];
>> H2 = cnotch('gaussian','reject', M, N, C2, 5);
>> % Compute the spectrum of the filtered transform and show
>> % it as an image.
>> P2 = gscale(fftshift(H2).*(tofloat(S)));
>> figure, imshow(P2)
>> % Filter image.
>> g2 = dftfilt(f,H2);
>> g2 = revertclass(g2);
>> figure, imshow(g2)
```

Figure 4.21(e) shows the notch filters superimposed on the spectrum and Fig. 4.21(f) is the filtered result. Comparing this image with Fig. 4.21(d) we see a reduction of high-frequency interference. Although this final result is far from perfect, it is a significant improvement over the original image. Considering the low resolution and significant corruption of this image, the result in Fig. 4.21(f) is as good as we can reasonably expect. ■

A special case of notch filtering involves filtering ranges of values along the axes of the DFT. The following function uses rectangles placed on the axes to achieve this. We show only the help text. See Appendix C for a complete listing of the code.

```
>> help recnotch

%RECNOTCH Generates rectangular notch (axes) filters.
%   H = RECNOTCH(NOTCH, MODE, M, N, W, SV, SH) generates an M-by-N
%   notch filter consisting of symmetric pairs of rectangles of
%   width W placed on the vertical and horizontal axes of the
%   (centered) frequency rectangle. The vertical rectangles start at
%   +SV and -SV on the vertical axis and extend to both ends of
%   the axis. Horizontal rectangles similarly start at +SH and -SH
```

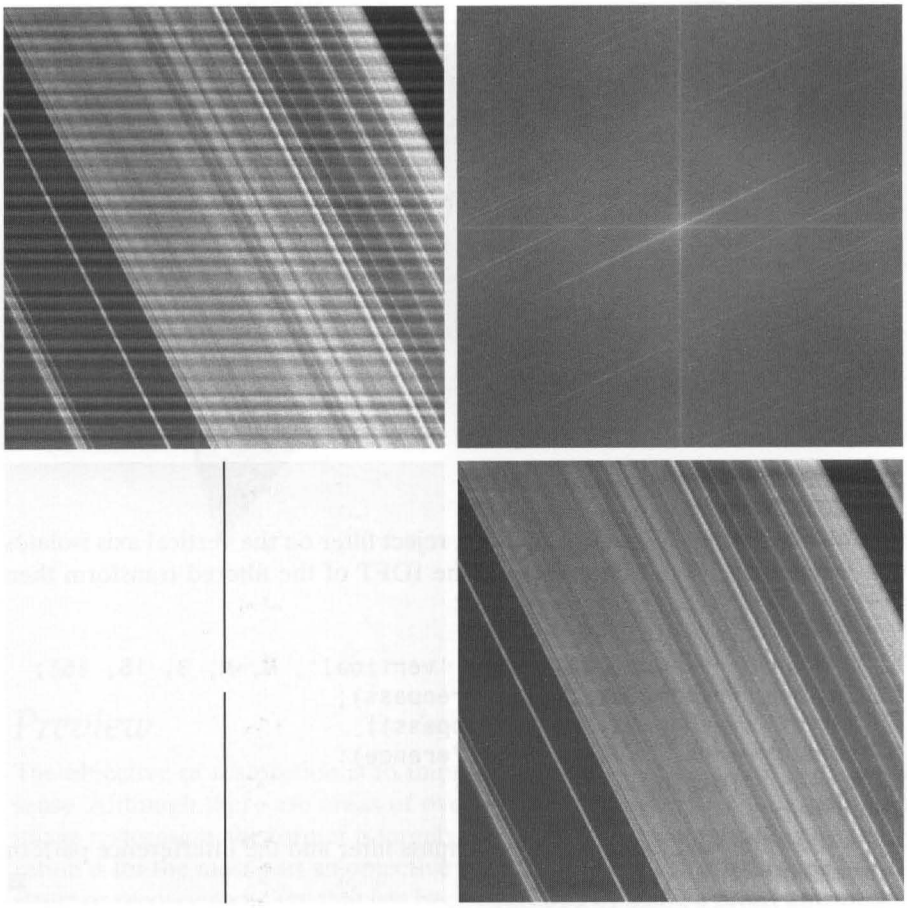
recnotch

```
%   and extend to both ends of the axis. These values are with
%   respect to the origin of the axes of the centered frequency
%   rectangle. For example, specifying SV = 50 creates a rectangle
%   of width W that starts 50 pixels above the center of the
%   vertical axis and extends up to the first row of the filter. A
%   similar rectangle is created starting 50 pixels below the center
%   and extending to the last row. W must be an odd number to
%   preserve the symmetry of the filtered Fourier transform.
%
%       Valid values of NOTCH are:
%
%           'reject'    Notchreject filter.
%
%           'pass'      Notchpass filter.
%
%
%       Valid values of MODE are:
%
%           'both'          Filtering on both axes.
%
%           'horizontal'    Filtering on horizontal axis only.
%
%           'vertical'      Filtering on vertical axis only.
%
%       One of these three values must be specified in the call.
%
%   H = RECNOTCH(NOTCH, MODE, M, N) sets W = 1, and SV = SH = 1.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).
```

EXAMPLE 4.10: Using notch filtering to reduce periodic interference caused by malfunctioning imaging equipment.

■ An important applications of notch filtering is in reducing periodic interference caused by malfunctioning imaging systems. Figure 4.22(a) shows a typical example. This is an image of the outer rings of planet Saturn, captured by *Cassini*, the first spacecraft to enter the planet's orbit. The horizontal bands are periodic interference caused an AC signal superimposed on the camera video signal just prior to digitizing the image. This was an unexpected problem that corrupted numerous images from the mission. Fortunately, this type of interference can be corrected by postprocessing, using methods such as those discussed in this section. Considering the cost and importance of these images, an "after-the-fact" solution to the interference problem is yet another example of the value and scope of image processing technology.

Figure 4.22(b) shows the Fourier spectrum. Because the interference is nearly periodic with respect to the vertical direction, we would expect to find energy bursts to be present in the vertical axis of the spectrum. Careful analysis of the spectrum indicates that indeed this is the case. We eliminate the source of interference by placing a narrow, rectangular notch filter on the vertical axis using the following commands:



a b
c d

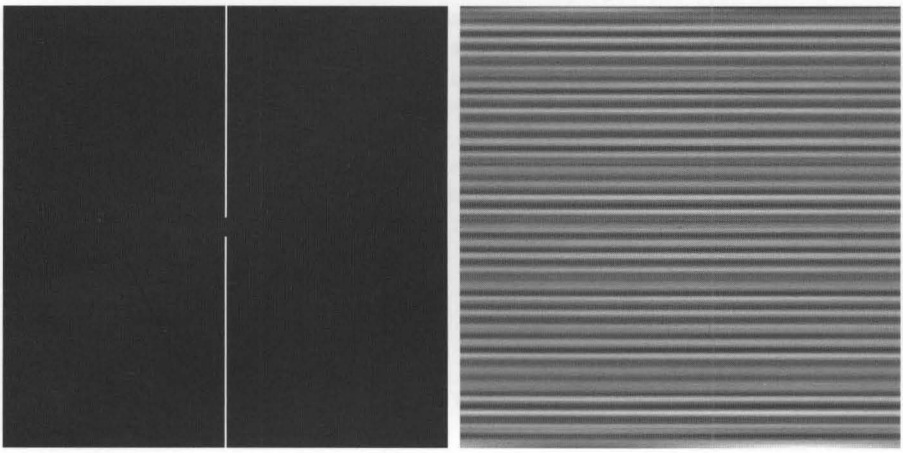
FIGURE 4.22
(a) 674×674 image of the Saturn rings, corrupted by periodic interference. (b) Spectrum: The bursts of energy on the vertical axis are caused by the interference. (c) Result of multiplying the DFT by a notch reject filter. (d) Result of computing the IDFT of (c). Note the improvement over (a). (Original image courtesy of Dr. Robert A. West, NASA/JPL.)

```
>> [M,N] = size(f);
>> [f, revertclass] = tofloat(f);
>> F = fft2(f);
>> S = gscale(log(1 + abs(fftshift(F))));
>> imshow(S);
>> H = recnotch('reject', 'vertical', M, N, 3, 15, 15);
>> figure, imshow(fftshift(H))
```

Figure 4.22(c) is the notch filter, and Fig. 4.22(d) shows the result of filtering:

```
>> g = dftfilt(f, H);
>> g = revertclass(g);
>> figure, imshow(g)
```

As you can see, Fig. 4.22(d) is a significant improvement over the original.



a b

FIGURE 4.23
(a) Notchpass filter. (b) Spatial interference pattern obtained by notchpass filtering.

Using a notchpass filter instead of a reject filter on the vertical axis isolates the frequencies of the interference. The IDFT of the filtered transform then yields the interference pattern itself:

```
>> Hrecpass = recnotch('pass', 'vertical', M, N, 3, 15, 15);
>> interference = dftfilt(f, Hrecpass);
>> figure, imshow(fftshift(Hrecpass))
>> interference = gscale(interference);
>> figure, imshow(interference)
```

Figures 4.23(a) and (b) show the notchpass filter and the interference pattern, respectively. ■

Summary

The material in this chapter is the foundation for using MATLAB and the Image Processing Toolbox in applications involving filtering in the frequency domain. In addition to the numerous image enhancement examples given in the preceding sections, frequency domain techniques play a fundamental role in image restoration (Chapter 5), image compression (Chapter 9), image segmentation (Chapter 11), and image description (Chapter 12).



5 Image Restoration and Reconstruction

Preview

The objective of restoration is to improve a given image in some predefined sense. Although there are areas of overlap between image enhancement and image restoration, the former is largely a subjective process, while image restoration is for the most part an objective process. Restoration attempts to reconstruct or recover an image that has been degraded by using a priori knowledge of the degradation phenomenon. Thus, restoration techniques are oriented toward modeling the degradation and applying the inverse process in order to recover the original image.

This approach usually involves formulating a criterion of goodness that yields an optimal estimate of the desired result. By contrast, enhancement techniques basically are heuristic procedures designed to manipulate an image in order to take advantage of the psychophysical aspects of the human visual system. For example, contrast stretching is considered an enhancement technique because it is based primarily on the pleasing aspects it might present to the viewer, whereas removal of image blur by applying a deblurring function is considered a restoration technique.

In this chapter we explore how to use MATLAB and Image Processing Toolbox capabilities to model degradation phenomena and to formulate restoration solutions. As in Chapters 3 and 4, some restoration techniques are best formulated in the spatial domain, while others are better suited for the frequency domain. Both methods are investigated in the sections that follow. We conclude the chapter with a discussion on the Radon transform and its use for image reconstruction from projections.

5.1 A Model of the Image Degradation/Restoration Process

As Fig. 5.1 shows, the degradation process is modeled in this chapter as a degradation function that, together with an additive noise term, operates on an input image $f(x, y)$ to produce a degraded image $g(x, y)$:

$$g(x, y) = H[f(x, y)] + \eta(x, y)$$

Given $g(x, y)$, some knowledge about the degradation function H, and some knowledge about the additive noise term $\eta(x, y)$, the objective of restoration is to obtain an estimate, $\hat{f}(x, y)$, of the original image. We want the estimate to be as close as possible to the original input image. In general, the more we know about H and $\eta(x, y)$, the closer $\hat{f}(x, y)$ will be to $f(x, y)$.

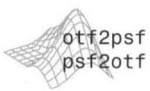
If H is a *linear, spatially invariant* process, it can be shown that the degraded image is given in the *spatial domain* by

$$g(x, y) = h(x, y) \star f(x, y) + \eta(x, y)$$

where $h(x, y)$ is the spatial representation of the degradation function and, as in Chapter 3, the symbol "$\star$" indicates convolution. We know from the discussion in Section 4.3.1 that convolution in the spatial domain and multiplication in the frequency domain constitute a Fourier transform pair, so we can write the preceding model in an equivalent *frequency domain* representation:

$$G(u, v) = H(u, v)F(u, v) + N(u, v)$$

where the terms in capital letters are the Fourier transforms of the corresponding terms in the spatial domain. The degradation function $F(u, v)$ sometimes is called the *optical transfer function* (OTF), a term derived from the Fourier analysis of optical systems. In the spatial domain, $h(x, y)$ is referred to as the *point spread function* (PSF), a term that arises from letting $h(x, y)$ operate on a point of light to obtain the characteristics of the degradation for any type of input. The OTF and PSF are a Fourier transform pair, and the toolbox provides two functions, `otf2psf` and `psf2otf`, for converting between them.



Because the degradation due to a linear, space-invariant degradation function, H, can be modeled as convolution, sometimes the degradation process is referred to as "convolving the image with a PSF." Similarly, the restoration process is sometimes referred to as *deconvolution*.

In the following three sections, we assume that H is the identity operator, and we deal only with degradation due to noise. Beginning in Section 5.6 we look at several methods for image restoration in the presence of both H and η.

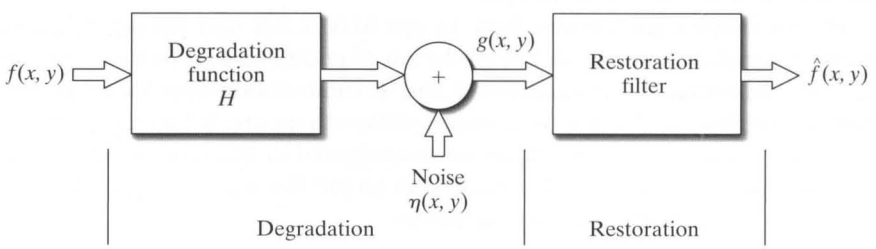


FIGURE 5.1
A model of the image degradation/restoration process.

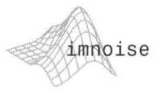
5.2 Noise Models

The ability to simulate the behavior and effects of noise is central to image restoration. In this chapter, we are interested in two basic types of noise models: noise in the spatial domain (described by the noise probability density function), and noise in the frequency domain, described by various Fourier properties of the noise. With the exception of the material in Section 5.2.3, we assume in this chapter that noise is independent of image coordinates.

5.2.1 Adding Noise to Images with Function `imnoise`

The Image Processing Toolbox uses function `imnoise` to corrupt an image with noise. This function has the basic syntax

```
g = imnoise(f, type, parameters)
```



where `f` is the input image, and `type` and `parameters` are as explained below. Function `imnoise` converts the input image to class `double` in the range [0, 1] before adding noise to it. This must be taken into account when specifying noise parameters. For example, to add Gaussian noise of mean 64 and variance 400 to a `uint8` image, we scale the mean to 64/255 and the variance to $400/(255)^2$ for input into `imnoise`. The syntax forms for this function are:

- `g = imnoise(f, 'gaussian', m, var)` adds Gaussian noise of mean `m` and variance `var` to image `f`. The default is zero mean noise with 0.01 variance.
- `g = imnoise(f, 'localvar', V)` adds zero-mean, Gaussian noise with local variance `V` to image `f`, where `V` is an array of the same size as `f` containing the desired variance values at each point.
- `g = imnoise(f, 'localvar', image_intensity, var)` adds zero-mean, Gaussian noise to image `f`, where the local variance of the noise, `var`, is a function of the image intensity values in `f`. The `image_intensity` and `var` arguments are vectors of the same size, and `plot(image_intensity, var)` plots the functional relationship between noise variance and image intensity. The `image_intensity` vector must contain normalized intensity values in the range [0, 1].
- `g = imnoise(f, 'salt & pepper', d)` corrupts image `f` with salt and pepper noise, where `d` is the noise density (i.e., the percent of the image area containing noise values). Thus, approximately `d*numel(f)` pixels are affected. The default is 0.05 noise density.
- `g = imnoise(f, 'speckle', var)` adds multiplicative noise to image `f`, using the equation `g = f + n.*f`, where `n` is uniformly distributed random noise with mean 0 and variance `var`. The default value of `var` is 0.04.
- `g = imnoise(f, 'poisson')` generates Poisson noise from the data instead of adding artificial noise to the data. In order to comply with Poisson statistics, the intensities of `uint8` and `uint16` images must correspond to the number of photons (or any other quanta of information). Double-precision images are used when the number of photons per pixel is larger than 65535 (but less

than 10^{12}). The intensity values vary between 0 and 1 and correspond to the number of photons divided by 10^{12}.

The following sections illustrate various uses of function `imnoise`.

5.2.2 Generating Spatial Random Noise with a Specified Distribution

Often, it is necessary to be able to generate noise of types and parameters beyond those available in function `imnoise`. Spatial noise values are random numbers, characterized by a probability density function (PDF) or, equivalently, by the corresponding cumulative distribution function (CDF). Random number generation for the types of distributions in which we are interested follow some fairly simple rules from probability theory.

Numerous random number generators are based on expressing the generation problem in terms of random numbers with a uniform CDF in the interval (0, 1). In some instances, the base random number generator of choice is a generator of Gaussian random numbers with zero mean and unit variance. Although we can generate these two types of noise using `imnoise`, it is simpler in the present context to use MATLAB function `rand` for uniform random numbers and `randn` for normal (Gaussian) random numbers. These functions are explained later in this section.

The foundation of the approach described in this section is a well-known result from probability (Peebles [1993]) which states that, if w is a uniformly distributed random variable in the interval (0, 1), then we can obtain a random variable z with a specified CDF, F, by solving the equation

$$z = F^{-1}(w)$$

This simple, yet powerful result can be stated equivalently as finding a solution to the equation $F(z) = w$.

EXAMPLE 5.1: Using uniform random numbers to generate random numbers with a specified distribution.

■ Assume that we have a generator of uniform random numbers, w, in the interval (0, 1), and suppose that we want to use it to generate random numbers, z, with a Rayleigh CDF, which has the form

$$F(z) = \begin{cases} 1 - e^{-(z-a)^2/b} & \text{for } z \geq a \\ 0 & \text{for } z < a \end{cases}$$

where $b > 0$. To find z we solve the equation

$$1 - e^{-(z-a)^2/b} = w$$

or

$$z = a + \sqrt{-b \ln(1 - w)}$$

Because the square root term is nonnegative, we are assured that no values of z less than a are generated, as required by the definition of the Rayleigh CDF.

Thus, a uniform random number w from our generator can be used in the previous equation to generate a random variable z having a Rayleigh distribution with parameters a and b.

In MATLAB this result is easily generalized to an array, `R`, of random numbers by using the expression

```
>> R = a + sqrt(b*log(1 - rand(M, N)));
```

where, as discussed in Section 3.2.2, `log` is the natural logarithm and, as explained later in this section, `rand` generates uniformly distributed random numbers in the interval (0, 1). If we let `M = N= 1`, then the preceding MATLAB command line yields a single value from a random variable with a Rayleigh distribution characterized by parameters a and b. ■

The expression $z = a + \sqrt{-b\ln(1 - w)}$ sometimes is called a *random number generator equation* because it establishes how to generate the desired random numbers. In this particular case, we were able to find a closed-form solution. As will be shown shortly, this is not always possible and the problem then becomes one of finding an applicable random number generator equation whose outputs will approximate random numbers with the specified CDF.

Table 5.1 lists the random variables of interest in the present discussion, along with their PDFs, CDFs, and random number generator equations. In some cases, as with the Rayleigh and exponential variables, it is possible to find a closed-form solution for the CDF and its inverse. This allows us to write an expression for the random number generator in terms of uniform random numbers, as illustrated in Example 5.1. In others, as in the case of the Gaussian and lognormal densities, closed-form solutions for the CDF do not exist, and it becomes necessary to find alternate ways to generate the desired random numbers. In the lognormal case, for instance, we make use of the knowledge that a lognormal random variable, z, is such that $\ln(z)$ has a Gaussian distribution; this allows us to write the expression shown in Table 5.1 in terms of Gaussian random variables with zero mean and unit variance. In other cases, it is advantageous to reformulate the problem to obtain an easier solution. For example, it can be shown that Erlang random numbers with parameters a and b can be obtained by adding b exponentially distributed random numbers that have parameter a (Leon-Garcia [1994]).

The random number generators available in `imnoise` and those shown in Table 5.1 play an important role in modeling the behavior of random noise in image-processing applications. We already saw the usefulness of the uniform distribution for generating random numbers with various CDFs. Gaussian noise is used as an approximation in cases such as imaging sensors operating at low light levels. Salt-and-pepper noise arises in faulty switching devices. The size of silver particles in a photographic emulsion is a random variable described by a lognormal distribution. Rayleigh noise arises in range imaging, while exponential and Erlang noise are useful in describing noise in laser imaging.

Unlike the other types of noise in Table 5.1, salt-and-pepper noise typically is viewed as generating an image with three values which, when working

TABLE 5.1 Generation of random variables.

Name	PDF	Mean and Variance	CDF	Generator[†]
Uniform	$p(z)=\begin{cases}\dfrac{1}{b-a} & \text{if } 0\le z\le b\\ 0 & \text{otherwise}\end{cases}$	$m=\dfrac{a+b}{2},\ \sigma^2=\dfrac{(b-a)^2}{12}$	$F(z)=\begin{cases}0 & z<a\\ \dfrac{z-a}{b-a} & a\le z\le b\\ 1 & z>b\end{cases}$	MATLAB function `rand`.
Gaussian	$p(z)=\dfrac{1}{\sqrt{2\pi}b}e^{-(z-a)^2/2b^2}$ $-\infty<z<\infty$	$m=a,\ \sigma^2=b^2$	$F(z)=\displaystyle\int_{-\infty}^{z}p(v)dv$	MATLAB function `randn`.
Lognormal	$p(z)=\dfrac{1}{\sqrt{2\pi}bz}e^{-[\ln(z)-a]^2/2b^2}$ $z>0$	$m=e^{a+(b^2/2)},\ \sigma^2=[e^{b^2}-1]e^{2a+b^2}$	$F(z)=\displaystyle\int_{0}^{z}p(v)dv$	$z=e^{bN(0,1)+a}$
Rayleigh	$p(z)=\begin{cases}\dfrac{2}{b}(z-a)e^{-(z-a)^2/b} & z\ge a\\ 0 & z<a\end{cases}$	$m=a+\sqrt{\pi b/4},\ \sigma^2=\dfrac{b(4-\pi)}{4}$	$F(z)=\begin{cases}1-e^{-(z-a)^2/b} & z\ge a\\ 0 & z<a\end{cases}$	$z=a+\sqrt{-b\ln[1-U(0,1)]}$
Exponential	$p(z)=\begin{cases}ae^{-az} & z\ge 0\\ 0 & z<0\end{cases}$	$m=\dfrac{1}{a},\ \sigma^2=\dfrac{1}{a^2}$	$F(z)=\begin{cases}1-e^{-az} & z\ge 0\\ 0 & z<0\end{cases}$	$z=-\dfrac{1}{a}\ln[1-U(0,1)]$
Erlang	$p(z)=\dfrac{a^b z^{b-1}}{(b-1)!}e^{-az}$ $z\ge 0$	$m=\dfrac{b}{a},\ \sigma^2=\dfrac{b}{a^2}$	$F(z)=\left[1-e^{-az}\displaystyle\sum_{n=0}^{b-1}\frac{(az)^n}{n!}\right]$ $z\ge 0$	$z=E_1+E_2+\cdots+E_b$ (The E's are exponential random numbers with parameter a.)
Salt & Pepper[‡]	$p(z)=\begin{cases}P_p & \text{for } z=0 \text{ (pepper)}\\ P_s & \text{for } z=2^n-1 \text{ (salt)}\\ 1-(P_p+P_s) & \text{for } z=k\\ & (0<k<2^n-1)\end{cases}$	$m=(0)P_p+k(1-P_p-P_s)+(2^n-1)P_s$ $\sigma^2=(0-m)^2P_p+(k-m)^2(1-P_p-P_s)+(2^n-1-m)^2P_s$	$F(z)=\begin{cases}0 & \text{for } z<0\\ P_p & \text{for } 0\le z<k\\ 1-P_s & \text{for } k\le z<2^n-1\\ 1 & \text{for } 2^n-1\le z\end{cases}$	MATLAB function `rand` with some additional logic.

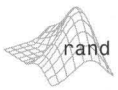
[†] $N(0,1)$ denotes normal (Gaussian) random numbers with mean 0 and variance 1. $U(0,1)$ denotes uniform random numbers in the range $(0,1)$.

[‡] As explained in the text, salt-and-pepper noise can be viewed as a random variable with three values, which in turn are used to modify the image to which noise is applied. In this sense, the mean and variance are not as meaningful as for the other noise types; we include them here for completeness (the 0s in the equation for the mean and variance are included to indicate explicitly that the intensity of pepper noise is assumed to be zero). Variable n is the number of bits in the digital image to which noise is applied.

with eight bits, are 0 with probability P_p, 255 with probability P_s, and k with probability $1 - (P_p + P_s)$, where k is any number between these two extremes. Let the noise image just described be denoted by $r(x, y)$. Then, we corrupt an image $f(x, y)$ [of the same size as $r(x, y)$] with salt and pepper noise by assigning a 0 to all locations in f where a 0 occurs in r. Similarly, we assign a 255 to all locations in f where 255 occurs in r. Finally, we leave unchanged in f all locations in which r contains the value k. The name *salt and pepper* arises from the fact that 0 is black and 255 is white in an 8-bit image. Although the preceding discussion was based on eight bits to simplify the explanation, it should be clear that the method is general and can be applied to any image with an arbitrary number of intensity levels, provided that we maintain two extreme values designated as salt and pepper. We could go one step further and, instead of two extreme values, we could generalize the previous discussion to two extreme *ranges* of values, although this is not typical in most applications.

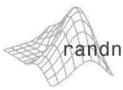
The probability, P, that a pixel is corrupted by salt-and-pepper noise is $P = P_p + P_s$. It is common terminology to refer to P as the *noise density*. If, for example, $P_p = 0.02$ and $P_s = 0.01$, we say that approximately 2% of the pixels in the image are corrupted by pepper noise, that 1% are corrupted by salt noise, and that the noise density is 0.03, meaning that a total of approximately 3% of the pixels in the image are corrupted by salt-and-pepper noise.

Custom M-function `imnoise2`, listed later in this section, generates random numbers having the CDFs in Table 5.1. This function uses MATLAB function `rand`, which has the syntax




```
A = rand(M, N)
```

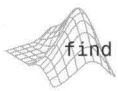
This function generates an array of size `M` × `N` whose entries are uniformly distributed numbers with values in the interval (0, 1). If `N` is omitted it defaults to `M`. If called without an argument, `rand` generates a single random number that changes each time the function is called. Similarly, the function




```
A = randn(M, N)
```

generates an `M` × `N` array whose elements are normal (Gaussian) numbers with zero mean and unit variance. If `N` is omitted it defaults to `M`. When called without an argument, `randn` generates a single random number.

Function `imnoise2` also uses MATLAB function `find`, which has the following syntax forms:




```
    I = find(A)
[r, c] = find(A)
[r, c, v] = find(A)
```

The first form returns in `I` the linear indices (see Section 2.8.5) of all the *nonzero* elements of `A`. If none is found, `find` returns an empty matrix. The second form returns the row and column indices of the nonzero entries in matrix `A`. In addition to returning the row and column indices, the third form also returns the nonzero values of `A` as a column vector, `v`.

The first form treats the array A in the format A(:), so I is a column vector. This form is quite useful in image processing. For example, to find and set to 0 all pixels in an image whose values are less than 128 we write

```
>> I = find(A < 128);
>> A(I) = 0;
```

This operation could be done also using logical indexing (see Section 2.8.4):

```
>> A(A < 128) = 0;
```

Recall that the logical statement A < 128 returns a 1 for the elements of A that satisfy the logical condition and 0 for those that do not. As another example, to set to 128 all pixels in the interval [64, 192] we write

```
>> I = find(A >= 64 & A <= 192)
>> A(I) = 128;
```

Equivalently, we could write

```
>> A(A >= 64 & A <= 192) = 128;
```

The type of indexing just discussed is used frequently in the remaining chapters of the book.

Unlike `imnoise`, the following M-function generates an $M \times N$ noise array, R, that is not scaled in any way. Another major difference is that `imnoise` outputs a noisy image, while `imnoise2` produces the noise pattern itself. The user specifies the desired values for the noise parameters directly. Note that the noise array resulting from salt-and-pepper noise has three values: 0 corresponding to pepper noise, 1 corresponding to salt noise, and 0.5 corresponding to no noise. This array needs to be processed further to make it useful. For example, to corrupt an image with this array, we find (using function `find` or the logical indexing illustrated above) all the coordinates in R that have value 0 and set the corresponding coordinates in the image to the smallest possible gray-level value (usually 0). Similarly, we find all the coordinates in R that have value 1 and set all the coordinates in the image to the highest possible value (usually 255 for an 8-bit image). All other pixels are left unchanged. This process simulates the manner in which salt-and-pepper noise affects an image.

Observe in the code for `imnoise2` how the `switch/case` statements are kept simple; that is, unless `case` computations can be implemented with one line, they are delegated to individual, separate functions appended at the end of the main program. This clarifies the logical flow of the code. Note also how all the defaults are handled by a separate function, `setDefaults`, which is also appended at the end of the main program. The objective is to modularize the code as much as possible for ease of interpretation and maintenance.

imnoise2

```
function R = imnoise2(type, varargin)
%IMNOISE2 Generates an array of random numbers with specified PDF.
%   R = IMNOISE2(TYPE, M, N, A, B) generates an array, R, of size
```

```
%   M-by-N, whose elements are random numbers of the specified TYPE
%   with parameters A and B. If only TYPE is included in the
%   input argument list, a single random number of the specified
%   TYPE and default parameters shown below is generated. If only
%   TYPE, M, and N are provided, the default parameters shown below
%   are used.  If M = N = 1, IMNOISE2 generates a single random
%   number of the specified TYPE and parameters A and B.
%
%   Valid values for TYPE and parameters A and B are:
%
%   'uniform'       Uniform random numbers in the interval (A, B).
%                   The default values are (0, 1).
%   'gaussian'      Gaussian random numbers with mean A and standard
%                   deviation B. The default values are A = 0,
%                   B = 1.
%   'salt & pepper' Salt and pepper numbers of amplitude 0 with
%                   probability Pa = A, and amplitude 1 with
%                   probability Pb = B. The default values are Pa =
%                   Pb = A = B = 0.05.  Note that the noise has
%                   values 0 (with probability Pa = A) and 1 (with
%                   probability Pb = B), so scaling is necessary if
%                   values other than 0 and 1 are required. The
%                   noise matrix R is assigned three values. If
%                   R(x, y) = 0, the noise at (x, y) is pepper
%                   (black). If R(x, y) = 1, the noise at (x, y) is
%                   salt (white). If R(x, y) = 0.5, there is no
%                   noise assigned to coordinates (x, y).
%   'lognormal'     Lognormal numbers with offset A and shape
%                   parameter B. The defaults are A = 1 and B =
%                   0.25.
%   'rayleigh'      Rayleigh noise with parameters A and B. The
%                   default values are A = 0 and B = 1.
%   'exponential'   Exponential random numbers with parameter A.
%                   The default is A = 1.
%   'erlang'        Erlang (gamma) random numbers with parameters A
%                   and B.  B must be a positive integer. The
%                   defaults are A = 2 and B = 5. Erlang random
%                   numbers are approximated as the sum of B
%                   exponential random numbers.

% Set defaults.
[M, N, a, b] = setDefaults(type, varargin{:});

% Begin processing. Use lower(type) to protect against input being
% capitalized.
switch lower(type)
case 'uniform'
   R = a + (b - a)*rand(M, N);
case 'gaussian'
   R = a + b*randn(M, N);
case 'salt & pepper'
   R = saltpepper(M, N, a, b);
```

```
case 'lognormal'
   R = exp(b*randn(M, N) + a);
case 'rayleigh'
   R = a + (-b*log(1 - rand(M, N))).^0.5;
case 'exponential'
   R = exponential(M, N, a);
case 'erlang'
   R = erlang(M, N, a, b);
otherwise
   error('Unknown distribution type.')
end

%-----------------------------------------------------------------
function R = saltpepper(M, N, a, b)
% Check to make sure that Pa + Pb is not > 1.
if (a + b) > 1
   error('The sum Pa + Pb must not exceed 1.')
end
R(1:M, 1:N) = 0.5;
% Generate an M-by-N array of uniformly-distributed random numbers
% in the range (0, 1). Then, Pa*(M*N) of them will have values <= a.
% The coordinates of these points we call 0 (pepper noise).
% Similarly, Pb*(M*N) points will have values in the range > a & <=
% (a + b). These we call 1 (salt noise).
X = rand(M, N);
R(X <= a) = 0;
u = a + b;
R(X > a & X <= u) = 1;

%-----------------------------------------------------------------
function R = exponential(M, N, a)
if a <= 0
   error('Parameter a must be positive for exponential type.')
end

k = -1/a;
R = k*log(1 - rand(M, N));

%-----------------------------------------------------------------
function R = erlang(M, N, a, b)
if (b ~= round(b) || b <= 0)
   error('Param b must be a positive integer for Erlang.')
end
k = -1/a;
R = zeros(M, N);
for j = 1:b
   R = R + k*log(1 - rand(M, N));
end

%-----------------------------------------------------------------
function varargout = setDefaults(type, varargin)
varargout = varargin;
```

```
P = numel(varargin);
if P < 4
   % Set default b.
   varargout{4} = 1;
end
if P < 3
   % Set default a.
   varargout{3} = 0;
end
if P < 2
   % Set default N.
   varargout{2} = 1;
end
if P < 1
   % Set default M.
   varargout{1} = 1;
end
if (P <= 2)
   switch type
      case 'salt & pepper'
         % a = b = 0.05.
         varargout{3} = 0.05;
         varargout{4} = 0.05;
      case 'lognormal'
         % a = 1; b = 0.25;
         varargout{3} = 1;
         varargout{4} = 0.25;
      case 'exponential'
         % a = 1.
         varargout{3} = 1;
      case 'erlang'
         % a = 2; b = 5.
         varargout{3} = 2;
         varargout{4} = 5;
   end
end
```

EXAMPLE 5.2: Histograms of data generated by function `imnoise2`.

■ Figure 5.2 shows histograms of all the random number types in Table 5.1. The data for each plot were generated using function `imnoise2`. For example, the data for Fig. 5.2(a) were generated by the following command:

```
>> r = imnoise2('gaussian', 100000, 1, 0, 1);
```

This statement generated a column vector, `r`, with 100000 elements, each being a random number from a Gaussian distribution with mean 0 and standard deviation of 1. A plot of the histogram was then obtained using function `hist`, which has the syntax

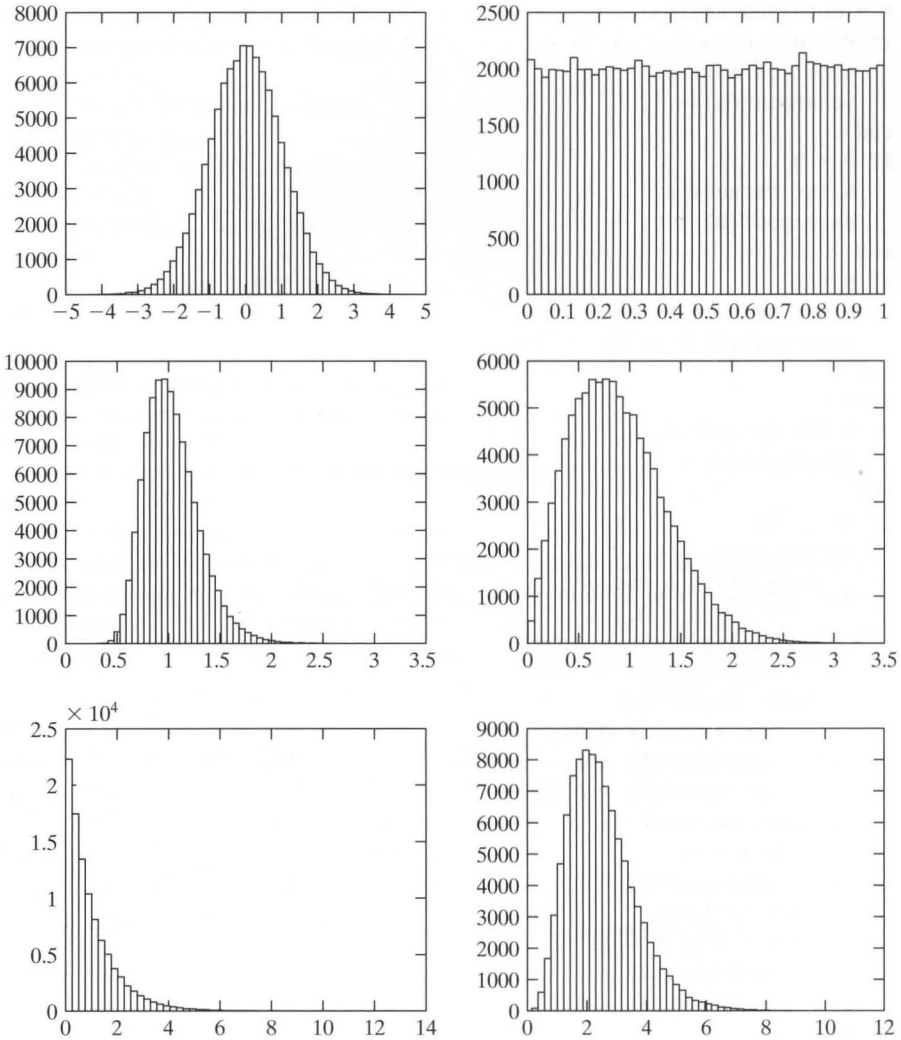


FIGURE 5.2 Histograms of random numbers: (a) Gaussian, (b) uniform, (c) lognormal, (d) Rayleigh, (e) exponential, and (f) Erlang. The default parameters listed in the explanation of function `imnoise2` were used in each case.



The syntax `h = hist(r, bins)` generates an array of size `1 x bins` containing the values of the histogram.

`hist(r, bins)`

where `bins` is the number of bins. We used `bins = 50` to generate the histograms in Fig. 5.2. The other histograms were generated in a similar manner. In each case, the parameters chosen were the default values listed in the explanation of function `imnoise2`. ■

5.2.3 Periodic Noise

Periodic noise in an image arises typically from electrical and/or electromechanical interference during image acquisition. This is the only type of spatially dependent noise that we consider in this chapter. As discussed in

Section 5.4, periodic noise typically is handled by filtering in the frequency domain. Our model of periodic noise is a 2-D, discrete sinusoid with equation

$$r(x, y) = A \sin\left[2\pi u_0(x + B_x)/M + 2\pi v_0(y + B_y)/N\right]$$

for $x = 0, 1, 2, \ldots, M - 1$ and $y = 0, 1, 2, \ldots, N - 1$, where A is the amplitude, u_0 and v_0 determine the sinusoidal frequencies with respect to the x- and y-axis, respectively, and B_x and B_y are phase displacements with respect to the origin. The DFT of this equation is

$$R(u, v) = j\frac{AMN}{2}\left[e^{-j2\pi(u_0B_x/M + v_0B_y/N)}\,\delta(u + u_0, v + v_0) - e^{j2\pi(u_0B_x/M + v_0B_y/N)}\,\delta(u - u_0, v - v_0)\right]$$

for $u = 0, 1, 2, \ldots, M - 1$ and $v = 0, 1, 2, \ldots, N - 1$, which we see is a pair of complex conjugate unit impulses located at $(u + u_0, v + v_0)$ and $(u - u_0, v - v_0)$, respectively. In other words, the first term inside the brackets in the preceding equation is zero unless $u = -u_0$ and $v = -v_0$, and the second is zero unless $u = u_0$ and $v = v_0$.

The following M-function accepts an arbitrary number of impulse locations (frequency coordinates), each with its own amplitude, frequency, and phase displacement parameters, and computes $r(x, y)$ as the sum of sinusoids of the form described in the previous paragraph. The function also outputs the Fourier transform, $R(u, v)$, of the sum of sinusoids, and the spectrum of $R(u, v)$. The sine waves are generated from the given impulse location information via the inverse DFT. This makes it more intuitive and simplifies visualization of frequency content in the spatial noise pattern. Only one pair of coordinates is required to define the location of an impulse. The program generates the conjugate symmetric impulses. Note in the code the use of function `ifftshift` to convert the centered `R` into the proper data arrangement for the `ifft2` operation, as discussed in Section 4.2.

imnoise3

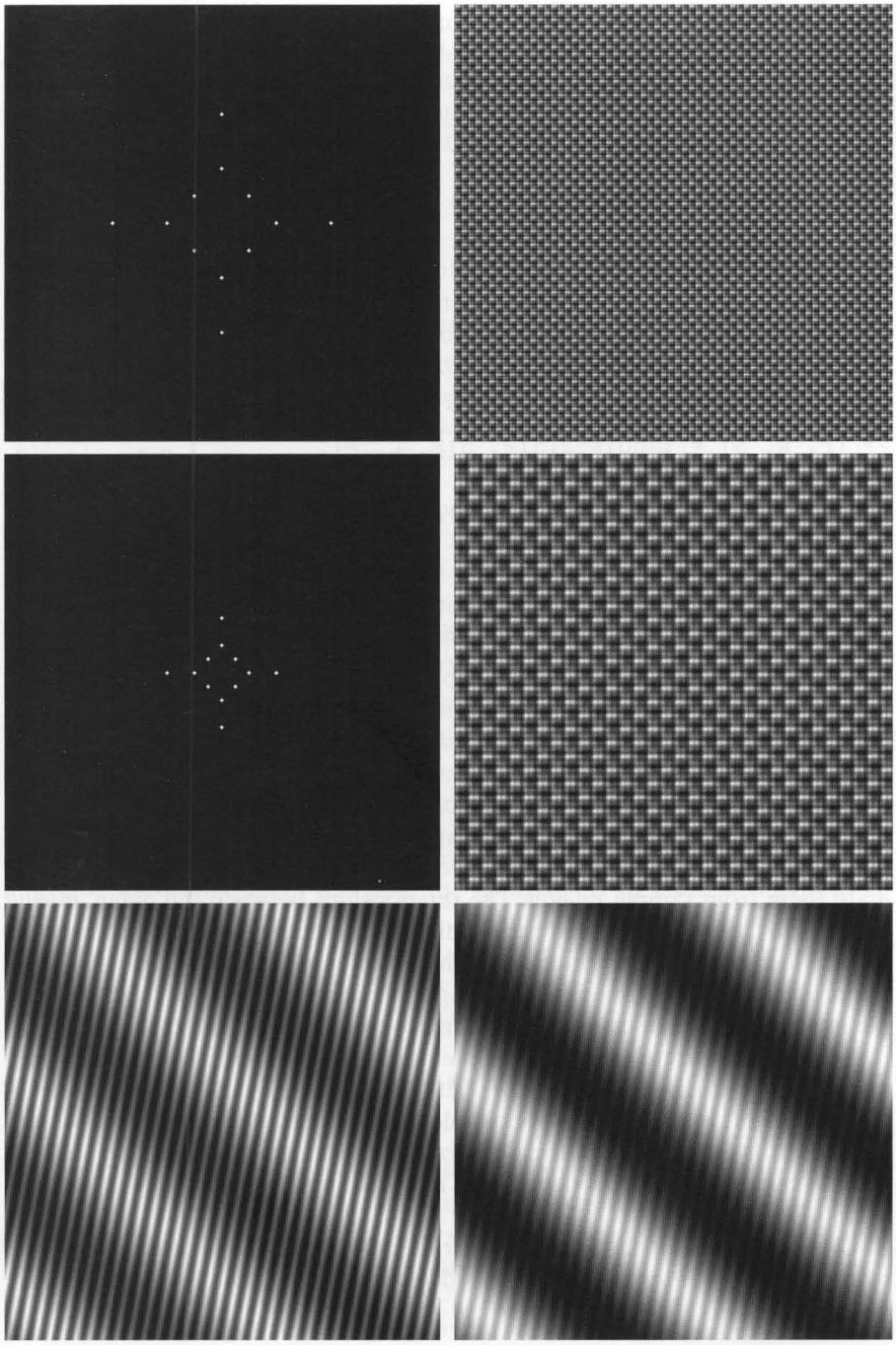
```
function [r, R, S] = imnoise3(M, N, C, A, B)
%IMNOISE3 Generates periodic noise.
%   [r, R, S] = IMNOISE3(M, N, C, A, B), generates a spatial
%   sinusoidal noise pattern, r, of size M-by-N, its Fourier
%   transform, R, and spectrum, S.  The remaining parameters are:
%
%   C is a K-by-2 matrix with K pairs of frequency domain
%   coordinates (u, v) that define the locations of impulses in the
%   frequency domain. The locations are with respect to the
%   frequency rectangle center at [floor(M/2) + 1, floor(N/2) + 1],
%   where the use of function floor is necessary to guarantee that
%   all values of (u, v) are integers, as required by all Fourier
%   formulations in the book. The impulse locations are specified as
%   integer increments with respect to the center. For example, if M =
%   N = 512, then the center is at (257, 257). To specify an
%   impulse at (280, 300) we specify the pair (23, 43); i.e., 257 +
```

```
%   23 = 280, and 257 + 43 = 300. Only one pair of coordinates is
%   required for each impulse. The conjugate pairs are generated
%   automatically.
%
%   A is a 1-by-K vector that contains the amplitude of each of the
%   K impulse pairs. If A is not included in the argument, the
%   default used is A = ONES(1, K).  B is then automatically set to
%   its default values (see next paragraph).  The value specified
%   for A(j) is associated with the coordinates in C(j, :).
%
%   B is a K-by-2 matrix containing the Bx and By phase components
%   for each impulse pair.  The default value for B is zeros(K, 2).

% Process input parameters.
K = size(C, 1);
if nargin < 4
   A = ones(1, K);
end
if nargin < 5
   B = zeros(K, 2);
end

% Generate R.
R = zeros(M, N);
for j = 1:K
   % Based on the equation for R(u, v), we know that the first term
   % of R(u, v) associated with a sinusoid is 0 unless u = -u0 and
   % v = -v0:
   u1 = floor(M/2) + 1 - C(j, 1);
   v1 = floor(N/2) + 1 - C(j, 2);
   R(u1, v1) = i*M*N*(A(j)/2) * exp(-i*2*pi*(C(j, 1)*B(j, 1)/M ...
                              + C(j, 2)*B(j, 2)/N));
   % Conjugate. The second term is zero unless u = u0 and v = v0:
   u2 = floor(M/2) + 1 + C(j, 1);
   v2 = floor(N/2) + 1 + C(j, 2);
   R(u2, v2) = -i*M*N*(A(j)/2) * exp(i*2*pi*(C(j, 1)*B(j, 1)/M ...
                              + C(j, 2)*B(j, 2)/N));
end

% Compute the spectrum and spatial sinusoidal pattern.
S = abs(R);
r = real(ifft2(ifftshift(R)));
```

EXAMPLE 5.3: Using function `imnoise3`.

■ Figures 5.3(a) and (b) show the spectrum and spatial sine noise pattern generated using the following commands:

```
>> C = [0 64; 0 128; 32 32; 64 0; 128 0; -32 32];
>> [r, R, S] = imnoise3(512, 512, C);
>> imshow(S, [ ])
>> figure, imshow(r, [ ])
```



a b
c d
e f

FIGURE 5.3
(a) Spectrum of specified impulses. (b) Corresponding sine noise pattern in the spatial domain. (c) and (d) A similar sequence. (e) and (f) Two other noise patterns. The dots in (a) and (c) were enlarged to make them easier to see.

As mentioned in the comments of function `imnoise3`, the (u, v) coordinates of the impulses are specified with respect to the center of the frequency rectangle (see Section 4.2 for more details about the coordinates of this center point). Figures 5.3(c) and (d) show the result obtained by repeating the previous commands, but with

```
>> C = [0 32; 0 64; 16 16; 32 0; 64 0; -16 16];
```

Similarly, Fig. 5.3(e) was obtained with

```
>> C = [6 32; -2 2];
```

Figure 5.3(f) was generated with the same `C`, but using a nondefault amplitude vector:

```
>> A = [1 5];
>> [r, R, S] = imnoise3(512, 512, C, A);
```

As Fig. 5.3(f) shows, the lower-frequency sine wave dominates the image. This is as expected because its amplitude is five times the amplitude of the higher-frequency component. ■

5.2.4 Estimating Noise Parameters

The parameters of periodic noise typically are estimated by analyzing the Fourier spectrum. Periodic noise produces frequency spikes that often can be detected even by visual inspection. Automated analysis is possible when the noise spikes are sufficiently pronounced, or when some knowledge about the frequency of the interference is available.

In the case of noise in the spatial domain, the parameters of the PDF may be known partially from sensor specifications, but it may be necessary to estimate them from sample images. The relationships between the mean, m, and variance, σ^2, of the noise, and the parameters a and b required to completely specify the noise PDFs of interest in this chapter are listed in Table 5.1. Thus, the problem becomes one of estimating the mean and variance from the sample image(s) and then using these estimates to solve for a and b.

A normalized histogram is obtained by dividing each component of the histogram by the number of pixels in the image. The sum of all the components of a normalized histogram is 1.

Let z_i be a discrete random variable that denotes intensity levels in an image, and let $p(z_i), i = 0, 1, 2, \ldots, L-1$, be the corresponding normalized histogram, where L is the number of possible intensity values. A histogram component, $p(z_i)$, is an estimate of the probability of occurrence of intensity value z_i, and the histogram may be viewed as a discrete approximation of the intensity PDF.

One of the principal approaches for describing the shape of a histogram is to use its *central moments* (also called *moments about the mean*), which are defined as

$$\mu_n = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i)$$

where n is the moment *order*, and m is the mean:

$$m = \sum_{i=0}^{L-1} z_i p(z_i)$$

Because the histogram is assumed to be normalized, the sum of all its components is 1, so we see from the preceding equations that $\mu_0 = 1$ and $\mu_1 = 0$. The second moment,

$$\mu_2 = \sum_{i=0}^{L-1} (z_i - m)^2 p(z_i)$$

is the variance. In this chapter, we are interested only in the mean and variance. Higher-order moments are discussed in Chapter 12.

Function `statmoments` computes the mean and central moments up to order n, and returns them in row vector `v`. Because the moment of order 0 is always 1, and the moment of order 1 is always 0, `statmoments` ignores these two moments and instead lets `v(1)` = m and `v(k)` = μ_k for $k = 2, 3, \ldots, n$. The syntax is as follows (see Appendix C for the code):

```
[v, unv] = statmoments(p, n)
```

`statmoments`

where `p` is the histogram vector and `n` is the number of moments to compute. It is required that the number of components of `p` be equal to 2^8 for class `uint8` images, 2^{16} for class `uint16` images, and 2^8 or 2^{16} for images of class `single` or `double`. Output vector `v` contains the normalized moments. The function scales the random variable to the range [0, 1], so all the moments are in this range also. Vector `unv` contains the same moments as `v`, but computed with the data in its original range of values. For example, if `length(p) = 256`, and `v(1) = 0.5`, then `unv(1)` would have the value `127.5`, which is half of the range `[0, 255]`.

Often, noise parameters must be estimated directly from a given noisy image or set of images. In this case, the approach is to select a region in an image with as featureless a background as possible, so that the variability of intensity values in the region will be due primarily to noise. To select a region of interest (ROI) in MATLAB we use function `roipoly`, which generates a polygonal ROI. This function has the basic syntax

```
B = roipoly(f, c, r)
```



where `f` is the image of interest, and `c` and `r` are vectors of corresponding (sequential) column and row coordinates of the vertices of the polygon (note that columns are specified first). The origin of the coordinates of the vertices is at the top, left. The output, `B`, is a *binary* image the same size as `f` with 0s outside the ROI and 1s inside. Image `B` is used typically as a mask to limit operations to within the region of interest.

To specify a polygonal ROI interactively, we use the syntax

```
B = roipoly(f)
```

which displays the image `f` on the screen and lets the user specify a polygon using the mouse. If `f` is omitted, `roipoly` operates on the last image displayed. Table 5.2 lists the various interactive capabilities of function `roipoly`. When you are finished positioning and sizing the polygon, you can create the mask `B` by double-clicking, or by right-clicking inside the region and selecting **Create mask** from the context menu.

TABLE 5.2 Interactive options for function `roipoly`.

Interactive Behavior	Description
Closing the polygon	Use any of the following mechanisms: • Move the pointer over the starting vertex of the polygon. The pointer changes to a circle, ○. Click either mouse button. • Double-click the left mouse button. This action creates a vertex at the point under the mouse pointer and draws a straight line connecting this vertex with the initial vertex. • Right-click the mouse. This draws a line connecting the last vertex selected with the initial vertex; it does not create a new vertex at the point under the mouse.
Moving the polygon	Move the pointer inside the region. The pointer changes to a fleur shape, ✥. Click and drag the polygon over the image.
Deleting the polygon	Press **Backspace**, **Escape**, or **Delete**, or right-click inside the region and select **Cancel** from the context menu. (If you delete the ROI, the function returns empty values.)
Moving a vertex	Move the pointer over a vertex. The pointer changes to a circle, ○. Click and drag the vertex to a new position.
Adding a vertex	Move the pointer over an edge of the polygon and press the **A** key. The pointer changes shape to ✧. Click the left mouse button to create a new vertex at that point.
Deleting a vertex	Move the pointer over the vertex. The pointer changes to a circle, ○. Right-click and select **Delete vertex** from the context menu. Function `roipoly` draws a new straight line between the two vertices that were neighbors of the deleted vertex.
Setting a polygon color	Move the pointer anywhere inside the boundary of the region. The pointer changes to ✥. Click the right mouse button. Select **Set color** from the context menu.
Retrieving the coordinates of the vertices	Move the pointer inside the region. The pointer changes to ✥. Right-click and select **Copy position** from the context menu to copy the current position to the Clipboard. The position is an $n \times 2$ array, each row of which contains the column and row coordinates (in that order) of each vertex; n is the number of vertices. The origin of the coordinate system is at the top, left, of the image.

To obtain the binary image *and* a list of the polygon vertices, we use the syntax

```
[B, c, r] = roipoly(...)
```

where `roipoly(...)` indicates any valid syntax form for this function and, as before, `c` and `r` are the column and row coordinates of the vertices. This format is particularly useful when the ROI is specified interactively because it gives the coordinates of the polygon vertices for use in other programs or for later duplication of the same ROI.

The following function computes the histogram of an ROI whose vertices are specified by vectors `c` and `r`, as in the preceding discussion. Note the use of function `roipoly` within the program to duplicate the polygonal region defined by `c` and `r`.

histroi

```
function [p, npix] = histroi(f, c, r)
%HISTROI Computes the histogram of an ROI in an image.
%   [P, NPIX] = HISTROI(F, C, R) computes the histogram, P, of a
%   polygonal region of interest (ROI) in image F. The polygonal
%   region is defined by the column and row coordinates of its
%   vertices, which are specified (sequentially) in vectors C and R,
%   respectively. All pixels of F must be >= 0. Parameter NPIX is the
%   number of pixels in the polygonal region.

% Generate the binary mask image.
B = roipoly(f, c, r);

% Compute the histogram of the pixels in the ROI.
p = imhist(f(B));

% Obtain the number of pixels in the ROI if requested in the output.
if nargout > 1
   npix = sum(B(:));
end
```

EXAMPLE 5.4: Estimating noise parameters.

■ Figure 5.4(a) shows a noisy image, denoted by `f` in the following discussion. The objective of this example is to estimate the noise type and its parameters using the techniques just discussed. Figure 5.4(b) shows a mask, `B`, generated using the interactive command:

```
>> [B, c, r] = roipoly(f);
```

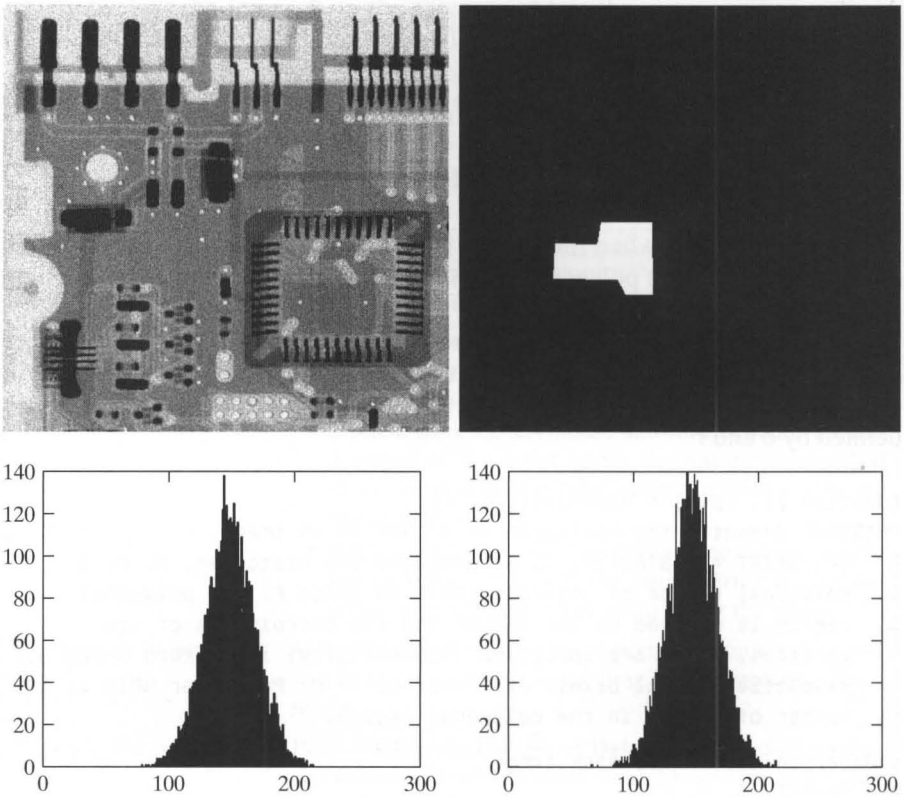
Figure 5.4(c) was generated using the commands

```
>> [h, npix] = histroi(f, c, r);
>> figure, bar(h, 1)
```

The mean and variance of the region masked by `B` were obtained as follows:

a b
c d

FIGURE 5.4
(a) Noisy image. (b) ROI generated interactively. (c) Histogram of ROI. (d) Histogram of Gaussian data generated using function `imnoise2`. (Original image courtesy of Lixi, Inc.)




```
>> [v, unv] = statmoments(h, 2);
>> v
v =
    0.5803      0.0063
>> unv
    147.9814    407.8679
```

It is evident from Fig. 5.4(c) that the noise is approximately Gaussian. By selecting an area of nearly constant background level (as we did here), and assuming that the noise is additive, we can estimate that the average intensity of the area in the ROI is reasonably close to the average gray level of the image in that area without noise, indicating that the noise in this case has zero mean. Also, the fact that the area has a nearly constant intensity level tells us that the variability in the region in the ROI is due primarily to the variance of the noise. (When feasible, another way to estimate the mean and variance of the noise is by imaging a target of constant, known reflectivity.) Figure 5.4(d) shows the histogram of a set of `npix` (this number is returned by `histroi`) Gaussian random variables using a mean of 147 and variance of 400 (approximately the values computed above), obtained with the following commands:

```
>> X = imnoise2('gaussian', npix, 1, 147, 20);
>> figure, hist(X, 130)
>> axis([0 300 0 140])
```

where the number of bins in `hist` was selected so that the result would be compatible with the plot in Fig. 5.4(c). The histogram in this figure was obtained within function `histroi` using `imhist`, which uses a different scaling than `hist`. We chose a set of `npix` random variables to generate `X`, so that the number of samples was the same in both histograms. The similarity between Figs. 5.4(c) and (d) clearly indicates that the noise is indeed well-approximated by a Gaussian distribution with parameters that are close to the estimates `v(1)` and `v(2)`. ■

5.3 Restoration in the Presence of Noise Only—Spatial Filtering

When the only degradation present is noise, it follows from the model in Section 5.1 that

$$g(x, y) = f(x, y) + \eta(x, y)$$

The method of choice for reducing noise in this case is spatial filtering, using the techniques developed in Sections 3.4 and 3.5. In this section we summarize and implement several spatial filters for noise reduction. Additional details on the characteristics of these filters are discussed in Gonzalez and Woods [2008].

5.3.1 Spatial Noise Filters

Table 5.3 lists the spatial filters of interest in this section, where S_{xy} denotes an $m \times n$ subimage (region) of the input noisy image, g. The subscripts on S indicate that the subimage is centered at coordinates (x, y) and $\hat{f}(x, y)$ (an estimate of f) denotes the filter response at those coordinates. The linear filters are implemented using function `imfilter` discussed in Section 3.4. The median, max, and min filters are nonlinear, order-statistic filters. The median filter can be implemented directly using toolbox function `medfilt2`. The max and min filters are implemented using functions `imdilate` and `imerode` discussed in Section 10.2.

The following custom function, which we call `spfilt`, performs filtering in the spatial domain using any of the filters listed in Table 5.3. Note the use of function `imlincomb` (mentioned in Section 2.10.2) to compute the linear combination of the inputs. Note also how function `tofloat` (see Section 2.7) is used to convert the output image to the same class as the input.

spfilt

```
function f = spfilt(g, type, varargin)
%SPFILT Performs linear and nonlinear spatial filtering.
%   F = SPFILT(G, TYPE, M, N, PARAMETER) performs spatial filtering
%   of image G using a TYPE filter of size M-by-N. Valid calls to
```

TABLE 5.3 Spatial filters. The variables m and n denote, respectively, the number of image rows and columns spanned by the filter.

Filter Name	Equation	Comments
Arithmetic mean	$\hat{f}(x,y) = \frac{1}{mn} \sum_{(s,t)\in S_{xy}} g(s,t)$	Implemented using toolbox functions `w = fspecial('average', [m, n])` and `f = imfilter(g, w)`.
Geometric mean	$\hat{f}(x,y) = \left[\prod_{(s,t)\in S_{xy}} g(s,t) \right]^{\frac{1}{mn}}$	This nonlinear filter is implemented using function `gmean` (see custom function `spfilt` in this section).
Harmonic mean	$\hat{f}(x,y) = \frac{mn}{\sum_{(s,t)\in S_{xy}} \frac{1}{g(s,t)}}$	This nonlinear filter is implemented using function `harmean` (see custom function `spfilt` in this section).
Contraharmonic mean	$\hat{f}(x,y) = \frac{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q+1}}{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q}}$	This nonlinear filter is implemented using function `charmean` (see custom function `spfilt` in this section).
Median	$\hat{f}(x,y) = \underset{(s,t)\in S_{xy}}{\text{median}}\{g(s,t)\}$	Implemented using toolbox function `medfilt2`: `f = medfilt2(g, [m n],'symmetric')`.
Max	$\hat{f}(x,y) = \max_{(s,t)\in S_{xy}} \{g(s,t)\}$	Implemented using toolbox function `imdilate`: `f = imdilate(g, ones(m, n))`.
Min	$\hat{f}(x,y) = \min_{(s,t)\in S_{xy}} \{g(s,t)\}$	Implemented using toolbox function `imerode`: `f = imerode(g, ones(m, n))`.
Midpoint	$\hat{f}(x,y) = \frac{1}{2}\left[\max_{(s,t)\in S_{xy}} \{g(s,t)\} + \min_{(s,t)\in S_{xy}} \{g(s,t)\} \right]$	Implemented as 0.5 times the sum of the max and min filtering results.
Alpha-trimmed mean	$\hat{f}(x,y) = \frac{1}{mn-d} \sum_{(s,t)\in S_{xy}} g_r(s,t)$	The $d/2$ lowest and $d/2$ highest pixels values of $g(s,t)$ in S_{xy} are deleted. Function $g_r(s,t)$ denotes the remaining $mn-d$ pixels in the neighborhood. Implemented using function `alphatrim` (see custom function `spfilt` in this section).

```
%   SPFILT are as follows:
%
%      F = SPFILT(G, 'amean', M, N)        Arithmetic mean filtering.
%      F = SPFILT(G, 'gmean', M, N)        Geometric mean filtering.
%      F = SPFILT(G, 'hmean', M, N)        Harmonic mean filtering.
%      F = SPFILT(G, 'chmean', M, N, Q)    Contraharmonic mean
%                                          filtering of order Q. The
%                                          default Q is 1.5.
%      F = SPFILT(G, 'median', M, N)       Median filtering.
%      F = SPFILT(G, 'max', M, N)          Max filtering.
%      F = SPFILT(G, 'min', M, N)          Min filtering.
%      F = SPFILT(G, 'midpoint', M, N)     Midpoint filtering.
%      F = SPFILT(G, 'atrimmed', M, N, D) Alpha-trimmed mean
%                                          filtering. Parameter D must
%                                          be a nonnegative even
%                                          integer; its default value
%                                          is 2.
%
%   The default values when only G and TYPE are input are M = N = 3,
%   Q = 1.5, and D = 2.

[m, n, Q, d] = processInputs(varargin{:});

% Do the filtering.
switch type
case 'amean'
   w = fspecial('average', [m n]);
   f = imfilter(g, w, 'replicate');
case 'gmean'
   f = gmean(g, m, n);
case 'hmean'
   f = harmean(g, m, n);
case 'chmean'
   f = charmean(g, m, n, Q);
case 'median'
   f = medfilt2(g, [m n], 'symmetric');
case 'max'
   f = imdilate(g, ones(m, n));
case 'min'
   f = imerode(g, ones(m, n));
case 'midpoint'
   f1 = ordfilt2(g, 1, ones(m, n), 'symmetric');
   f2 = ordfilt2(g, m*n, ones(m, n), 'symmetric');
   f = imlincomb(0.5, f1, 0.5, f2);
case 'atrimmed'
   f = alphatrim(g, m, n, d);
otherwise
   error('Unknown filter type.')
end
```

```
%-----------------------------------------------------------------------%
function f = gmean(g, m, n)
%  Implements a geometric mean filter.
[g, revertClass] = tofloat(g);
f = exp(imfilter(log(g), ones(m, n), 'replicate')).^(1 / m / n);
f = revertClass(f);

%-----------------------------------------------------------------------%
function f = harmean(g, m, n)
%  Implements a harmonic mean filter.
[g, revertClass] = tofloat(g);
f = m * n ./ imfilter(1./(g + eps),ones(m, n), 'replicate');
f = revertClass(f);

%-----------------------------------------------------------------------%
function f = charmean(g, m, n, q)
%  Implements a contraharmonic mean filter.
[g, revertClass] = tofloat(g);
f = imfilter(g.^(q+1), ones(m, n), 'replicate');
f = f ./ (imfilter(g.^q, ones(m, n), 'replicate') + eps);
f = revertClass(f);

%-----------------------------------------------------------------------%
function f = alphatrim(g, m, n, d)
%  Implements an alpha-trimmed mean filter.
if (d <= 0) || (d/2 ~= round(d/2))
   error('d must be a positive, even integer.')
end
[g, revertClass] = tofloat(g);
f = imfilter(g, ones(m, n), 'symmetric');
for k = 1:d/2
   f = f - ordfilt2(g, k, ones(m, n), 'symmetric');
end
for k = (m*n - (d/2) + 1):m*n
   f = f - ordfilt2(g, k, ones(m, n), 'symmetric');
end
f = f / (m*n - d);
f = revertClass(f);

%-----------------------------------------------------------------------%
function [m, n, Q, d] = processInputs(varargin)
m = 3;
n = 3;
Q = 1.5;
d = 2;
if nargin > 0
   m = varargin{1};
end
if nargin > 1
   n = varargin{2};
end
if nargin > 2
```

```
    Q = varargin{3};
    d = varargin{3};
end
```

EXAMPLE 5.5: Using function `spfilt`.

■ The image in Fig. 5.5(a) is a `uint8` image corrupted by pepper noise only with probability 0.1. This image was generated using the following commands [`f` is the image from Fig. 3.19(a)]:

```
>> [M, N] = size(f);
>> R = imnoise2('salt & pepper', M, N, 0.1, 0);
>> gp = f;
>> gp(R == 0) = 0;
```

The image in Fig. 5.5(b) was corrupted by salt noise only using the statements

```
>> R = imnoise2('salt & pepper', M, N, 0, 0.1);
>> gs = f;
>> gs(R == 1) = 255;
```

A good approach for filtering pepper noise is to use a contraharmonic filter with a positive value of Q. Figure 5.5(c) was generated using the statement

```
>> fp = spfilt(gp, 'chmean', 3, 3, 1.5);
```

Similarly, salt noise can be filtered using a contraharmonic filter with a negative value of Q:

```
>> fs = spfilt(gs, 'chmean', 3, 3, -1.5);
```

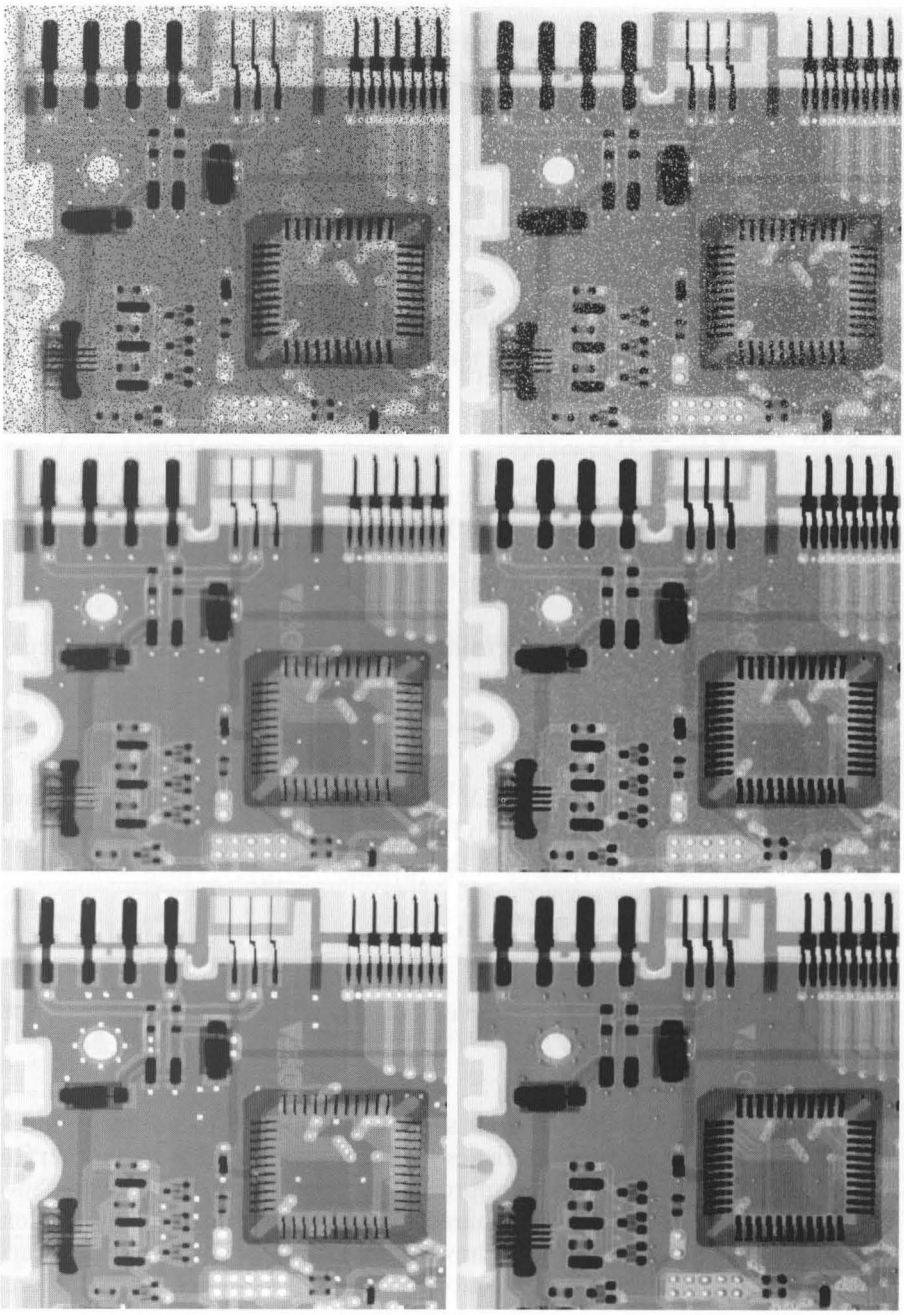
Figure 5.5(d) shows the result. Similar results can be obtained using max and min filters. For example, the images in Figs. 5.5(e) and (f) were generated from Figs. 5.5(a) and (b), respectively, with the following commands:

```
>> fpmax = spfilt(gp, 'max', 3, 3);
>> fsmin = spfilt(gs, 'min', 3, 3);
```

Other solutions using `spfilt` are implemented in a similar manner. ■

5.3.2 Adaptive Spatial Filters

The filters discussed in the previous section are applied to an image independently of how image characteristics vary from one location to another. In some applications, results can be improved by using filters capable of adapting their behavior based on the characteristics of the image in the region being filtered. As an illustration of how to implement adaptive spatial filters in MATLAB, we consider in this section an adaptive median filter. As before, S_{xy} denotes a subimage centered at location (x, y) in the image being processed. The algorithm, due to Eng and Ma [2001] and explained in detail in Gonzalez and Woods [2008], is as follows. Let



a b
c d
e f

FIGURE 5.5
(a) Image corrupted by pepper noise with probability 0.1. (b) Image corrupted by salt noise with the same probability. (c) Result of filtering (a) with a 3×3 contraharmonic filter of order $Q = 1.5$. (d) Result of filtering (b) with $Q = -1.5$. (e) Result of filtering (a) with a 3×3 max filter. (f) Result of filtering (b) with a 3×3 min filter.

$$z_{\min} = \text{minimum intensity value in } S_{xy}$$
$$z_{\max} = \text{maximum intensity value in } S_{xy}$$
$$z_{\text{med}} = \text{median of the intensity values in } S_{xy}$$
$$z_{xy} = \text{intensity value at coordinates } (x, y)$$

The adaptive median filtering algorithm uses two processing levels, denoted level A and level B:

Level A: If $z_{\min} < z_{\text{med}} < z_{\max}$, go to Level B
Else increase the window size
If window size $\leq S_{\max}$, repeat level A
Else output z_{med}

Level B: If $z_{\min} < z_{xy} < z_{\max}$, output z_{xy}
Else output z_{med}

where $S_{\max}$ denotes the maximum allowed size of the adaptive filter window. Another option in the last step in Level A is to output z_{xy} instead of the median. This produces a slightly less blurred result but can fail to detect salt (pepper) noise embedded in a constant background having the same value as pepper (salt) noise.

An M-function that implements this algorithm, which we call `adpmedian`, is included in Appendix C. The syntax is

```
f = adpmedian(g, Smax)
```

`adpmedian`

where `g` is the image to be filtered and, as defined above, `Smax` is the maximum allowed size of the adaptive filter window.

EXAMPLE 5.6: Adaptive median filtering.

■ Figure 5.6(a) shows the circuit board image, `f`, corrupted by salt-and-pepper noise generated using the command

```
>> g = imnoise(f, 'salt & pepper', .25);
```

and Fig. 5.6(b) shows the result obtained using the command

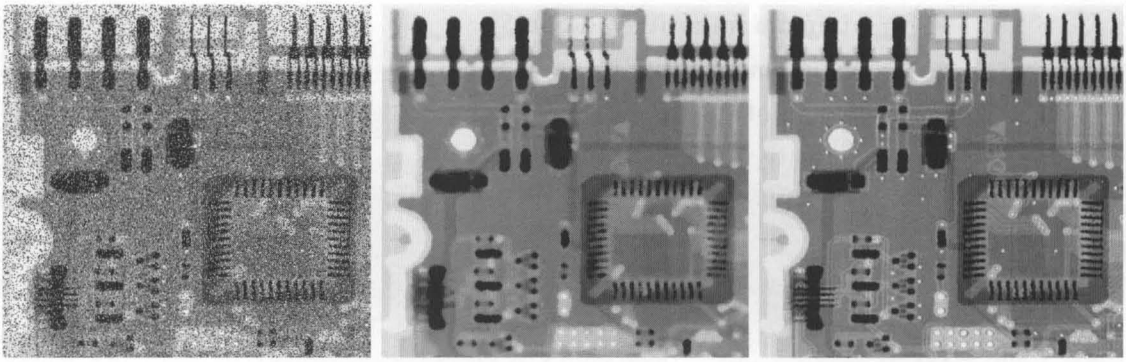
```
>> f1 = medfilt2(g, [7 7], 'symmetric');
```

See Section 3.5.2 regarding the use of function `medfilt2`.

This image is reasonably free of noise, but it is quite blurred and distorted (e.g., see the connector fingers in the top middle of the image). On the other hand, the command

```
>> f2 = adpmedian(g, 7);
```

yielded the image in Fig. 5.6(c), which is also reasonably free of noise, but is considerably less blurred and distorted than Fig. 5.6(b). ■



a b c

FIGURE 5.6 (a) Image corrupted by salt-and-pepper noise with density 0.25. (b) Result obtained using a median filter of size 7×7. (c) Result obtained using adaptive median filtering with $S_{max} = 7$.

5.4 Periodic Noise Reduction Using Frequency Domain Filtering

As noted in Section 5.2.3, periodic noise produces impulse-like bursts that often are visible in the Fourier spectrum. The principal approach for filtering these components is to use notchreject filtering. As discussed in Section 4.7.2, the general expression for a notchreject filter having Q notch pairs is

$$H_{NR}(u,v) = \prod_{k=1}^{Q} H_k(u,v) H_{-k}(u,v)$$

where $H_k(u,v)$ and $H_{-k}(u,v)$ are highpass filters with centers at (u_k, v_k) and $(-u_k, -v_k)$, respectively. These translated centers are specified with respect to the center of the frequency rectangle, $(M/2, N/2)$. Therefore, the distance computations for the filters are given by the expression

$$D_k(u,v) = \left[(u - M/2 - u_k)^2 + (v - N/2 - v_k)^2\right]^{\frac{1}{2}}$$

and

$$D_{-k}(u,v) = \left[(u - M/2 + u_k)^2 + (v - N/2 + v_k)^2\right]^{\frac{1}{2}}$$

We discuss several types of notchreject filters in Section 4.7.2 and give a custom function, `cnotch`, for generating these filters. A special case of notchreject filtering that notches out components along of the frequency axes also are used for image restoration. Function `recnotch` discussed in Section 4.7.2 implements this type of filter. Examples 4.9 and 4.10 demonstrate the effectiveness of notchreject filtering for periodic noise reduction.

5.5 Modeling the Degradation Function

When equipment similar to the equipment that generated a degraded image is available, it is possible sometimes to determine the nature of the degradation by experimenting with various equipment settings. However, relevant imaging equipment availability is the exception, rather than the rule, in the solution of image restoration problems, and a typical approach is to experiment by generating PSFs and testing the results with various restoration algorithms. Another approach is to attempt to model the PSF mathematically. This approach is outside the mainstream of our discussion here; for an introduction to this topic see Gonzalez and Woods [2008]. Finally, when no information is available about the PSF, we can resort to "blind deconvolution" for inferring the PSF. This approach is discussed in Section 5.10. The focus of the remainder of the present section is on various techniques for modeling PSFs by using functions `imfilter` and `fspecial`, introduced in Sections 3.4 and 3.5, respectively, and the various noise-generating functions discussed earlier in this chapter.

One of the principal degradations encountered in image restoration problems is image blur. Blur that occurs with the scene and sensor at rest with respect to each other can be modeled by spatial or frequency domain lowpass filters. Another important degradation model is image blur caused by uniform linear motion between the sensor and scene during image acquisition. Image blur can be modeled using toolbox function `fspecial`:

```
PSF = fspecial('motion', len, theta)
```

This call to `fspecial` returns a PSF that approximates the effects of linear motion of a camera by `len` pixels. Parameter `theta` is in degrees, measured with respect to the positive horizontal axis in a counter-clockwise direction. The default values of `len` and `theta` are 9 and 0, respectively. These settings correspond to motion of 9 pixels in the horizontal direction.

We use function `imfilter` to create a degraded image with a `PSF` that is either known or is computed by using the method just described:

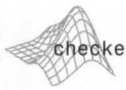
```
>> g = imfilter(f, PSF, 'circular');
```

where `'circular'` (Table 3.2) is used to reduce border effects. We then complete the degraded image model by adding noise, as appropriate:

```
>> g = g + noise;
```

where `noise` is a random noise image of the same size as `g`, generated using one of the methods discussed in Section 5.2.

When comparing the suitability of the various approaches discussed in this and the following sections, it is useful to use the same image or test pattern so that comparisons are meaningful. The test pattern generated by function `checkerboard` is particularly useful for this purpose because its size can be scaled without affecting its principal features. The syntax is




```
C = checkerboard(NP, M, N)
```

where NP is the number of pixels on the side of each square, M is the number of rows, and N is the number of columns. If N is omitted, it defaults to M. If both M and N are omitted, a square checkerboard with 8 squares on the side is generated. If, in addition, NP is omitted, it defaults to 10 pixels. The light squares on the left half of the checkerboard are white. The light squares on the right half of the checkerboard are gray. To generate a checkerboard in which all light squares are white we use the command

Using the > operator produces a `logical` result; `im2double` is used to produce an image of class `double`, which is consistent with the class of the output of function `checkerboard`.

```
>> K = checkerboard(NP, M, N) > 0.5;
```

The images generated by `checkerboard` are of class `double` with values in the range [0, 1].

Because some restoration algorithms are slow for large images, a good approach is to experiment with small images to reduce computation time. In this case, it is useful for display purposes to be able to zoom an image by pixel replication. The following function does this (see Appendix C for the code):

`pixeldup`

```
B = pixeldup(A, m, n)
```

This function duplicates every pixel in A a total of m times in the vertical direction and n times in the horizontal direction. If n is omitted, it defaults to m.

EXAMPLE 5.7: Modeling a blurred, noisy image.

■ Figure 5.7(a) shows a checkerboard image generated by the command

```
>> f = checkerboard(8); % Image is of class double.
```

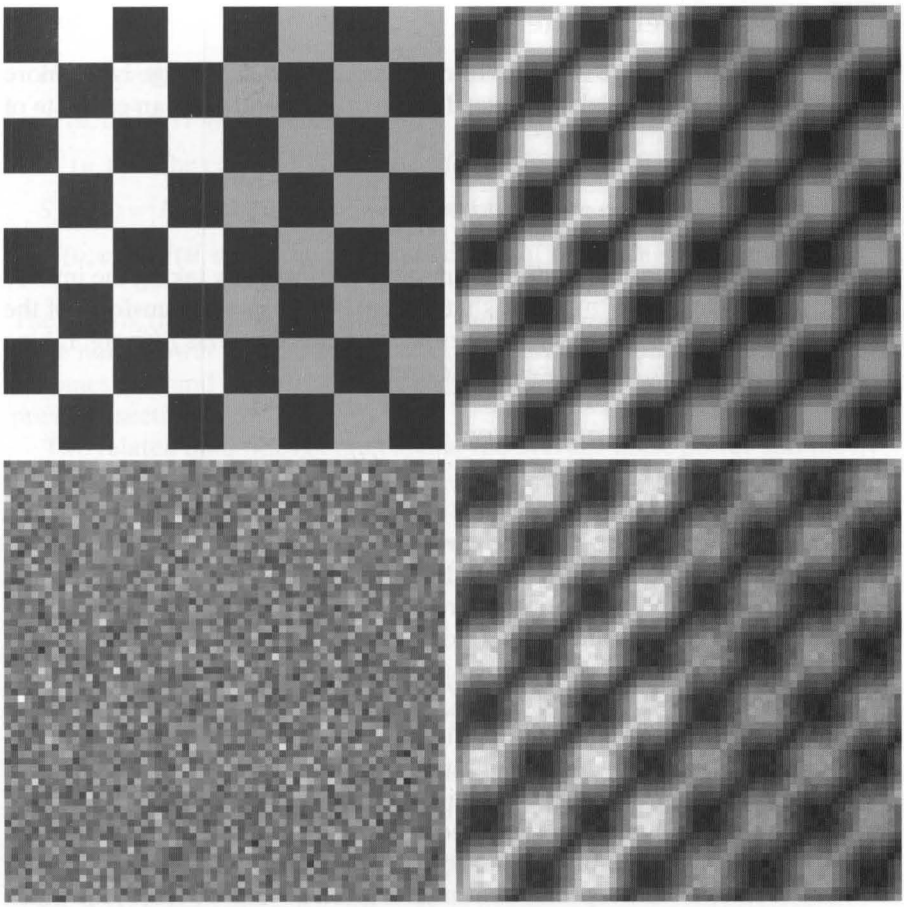
The degraded image in Fig. 5.7(b) was generated using the commands

```
>> PSF = fspecial('motion', 7, 45);
>> gb = imfilter(f, PSF, 'circular');
```

The PSF is a spatial filter. Its values are

```
>> PSF
PSF =
         0         0         0         0         0    0.0145         0
         0         0         0         0    0.0376    0.1283    0.0145
         0         0         0    0.0376    0.1283    0.0376         0
         0         0    0.0376    0.1283    0.0376         0         0
         0    0.0376    0.1283    0.0376         0         0         0
    0.0145    0.1283    0.0376         0         0         0         0
         0    0.0145         0         0         0         0         0
```

The noisy pattern in Fig. 5.7(c) is a Gaussian noise image with mean 0 and variance 0.001. It was generated using the command



a b
c d

FIGURE 5.7
(a) Original image. (b) Image blurred using `fspecial` with `len = 7`, and `theta = -45` degrees. (c) Noise image. (d) Sum of (b) and (c).

```
>> noise = imnoise2('Gaussian', size(f,1), size(f, 2), 0,...
                                sqrt(0.001));
```

The blurred noisy image in Fig. 5.7(d) was generated as

```
>> g = gb + noise;
```

The noise is not easily visible in this image because its maximum value is approximately 0.15, whereas the maximum value of the image is 1. As will be shown in Sections 5.7 and 5.8, however, this level of noise is not insignificant when attempting to restore g. Finally, we point out that all images in Fig. 5.7 were zoomed to size 512×512 and were displayed using a command of the form

```
>> imshow(pixeldup(f, 8), [ ])
```

The image in Fig. 5.7(d) is restored in Examples 5.8 and 5.9. ■

5.6 Direct Inverse Filtering

The simplest approach we can take to restoring a degraded image is to ignore the noise term in the model introduced in Section 5.1 and form an estimate of the form

$$\hat{F}(u,v) = \frac{G(u,v)}{H(u,v)}$$

Then, we obtain the corresponding estimate of the image by taking the inverse Fourier transform of $\hat{F}(u,v)$ [recall that $G(u,v)$ is the Fourier transform of the degraded image]. This approach is appropriately called *inverse filtering*. Taking noise into account, we can express our estimate as

$$\hat{F}(u,v) = F(u,v) + \frac{N(u,v)}{H(u,v)}$$

This deceptively simple expression tells us that, even if we knew $H(u,v)$ exactly, we could not recover $F(u,v)$ [and hence the original, undegraded image $f(x,y)$] because the noise component is a random function whose Fourier transform, $N(u,v)$, is not known. In addition, there usually is a problem in practice with function $H(u,v)$ having numerous zeros. Even if the noise term $N(u,v)$ were negligible, dividing it by vanishing values of $H(u,v)$ would dominate restoration estimates.

The typical approach when attempting inverse filtering is to form the ratio $\hat{F}(u,v) = G(u,v)/H(u,v)$ and then limit the frequency range for obtaining the inverse, to frequencies "near" the origin. The idea is that zeros in $H(u,v)$ are less likely to occur near the origin because the magnitude of the transform typically is at its highest values in that region. There are numerous variations of this basic theme, in which special treatment is given at values of (u,v) for which H is zero or near zero. This type of approach sometimes is called *pseudoinverse* filtering. In general, approaches based on inverse filtering of this type seldom are practical, as Example 5.8 in the next section shows.

5.7 Wiener Filtering

Wiener filtering (after N. Wiener, who first proposed the method in 1942) is one of the earliest and best known approaches to linear image restoration. A Wiener filter seeks an estimate $\hat{f}$ that minimizes the statistical error function

$$e^2 = E\left\{\left(f - \hat{f}\right)^2\right\}$$

where E is the expected value operator and f is the undegraded image. The solution to this expression in the frequency domain is

$$\hat{F}(u,v) = \left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}\right]G(u,v)$$

where

$H(u,v)$ = the degradation function

$|H(u,v)|^2 = H^*(u,v)H(u,v)$

$H^*(u,v)$ = the complex conjugate of $H(u,v)$

$S_\eta(u,v) = |N(u,v)|^2$ = the power spectrum of the noise

$S_f(u,v) = |F(u,v)|^2$ = the power spectrum of the undegraded image

The ratio $S_\eta(u,v)/S_f(u,v)$ is called the *noise-to-signal power ratio*. We see that if the noise power spectrum is zero for all relevant values of u and v, this ratio becomes zero and the Wiener filter reduces to the inverse filter discussed in the previous section.

Two related quantities of interest are the average noise power and the average image power, defined as

$$\eta_A = \frac{1}{MN}\sum_u\sum_v S_\eta(u,v)$$

and

$$f_A = \frac{1}{MN}\sum_u\sum_v S_f(u,v)$$

where, as usual, M and N denote the number of rows and columns of the image and noise arrays, respectively. These quantities are scalar constants, and their ratio,

$$R = \frac{\eta_A}{f_A}$$

which is also a scalar, is used sometimes to generate a constant array in place of the function $S_\eta(u,v)/S_f(u,v)$. In this case, even if the actual ratio is not known, it becomes a simple matter to experiment interactively by varying R and viewing the restored results. This, of course, is a crude approximation that assumes that the functions are constant. Replacing $S_\eta(u,v)/S_f(u,v)$ by a constant array in the preceding filter equation results in the so-called *parametric Wiener filter*. As illustrated in Example 5.8, even the simple act of using a constant array can yield significant improvements over direct inverse filtering.

Wiener filtering is implemented by the Image Processing Toolbox function `deconvwnr`, which has three possible syntax forms. In all three forms, `g` denotes the degraded image and `frest` is the restored image. The first syntax form,

```
frest = deconvwnr(g, PSF)
```



assumes that the noise-to-signal ratio is zero. Thus, this form of the Wiener filter is the inverse filter discussed in Section 5.6. The syntax

```
frest = deconvwnr(g, PSF, NSPR)
```

assumes that the noise-to-signal power ratio is known, either as a constant or as an array; the function accepts either one. This is the syntax used to implement the parametric Wiener filter, in which case NSPR would be a scalar input. Finally, the syntax

```
frest = deconvwnr(g, PSF, NACORR, FACORR)
```

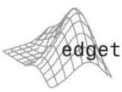
assumes that autocorrelation functions, NACORR and FACORR, of the noise and undegraded image are known. Note that this form of deconvwnr uses the autocorrelation of η and f instead of the power spectrum of these functions. From the correlation theorem we know that

See Gonzalez and Woods [2208] for a discussion of the correlation theorem.

$$|F(u,v)|^2 = \Im[f(x,y) \star f(x,y)]$$

where "$\star$" denotes the correlation operation and $\Im$ denotes the Fourier transform. This expression indicates that we can obtain the autocorrelation function, $f(x,y) \star f(x,y)$, for use in deconvwnr by computing the inverse Fourier transform of the power spectrum. Similar comments hold for the autocorrelation of the noise.

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it helps sometimes to use function edgetaper prior to calling deconvwnr. The syntax is




```
J = edgetaper(I, PSF)
```

This function blurs the edges of the input image, I, using the point spread function, PSF. The output image, J, is the weighted sum of I and its blurred version. The weighting array, determined by the autocorrelation function of PSF, makes J equal to I in its central region, and equal to the blurred version of I near the edges.

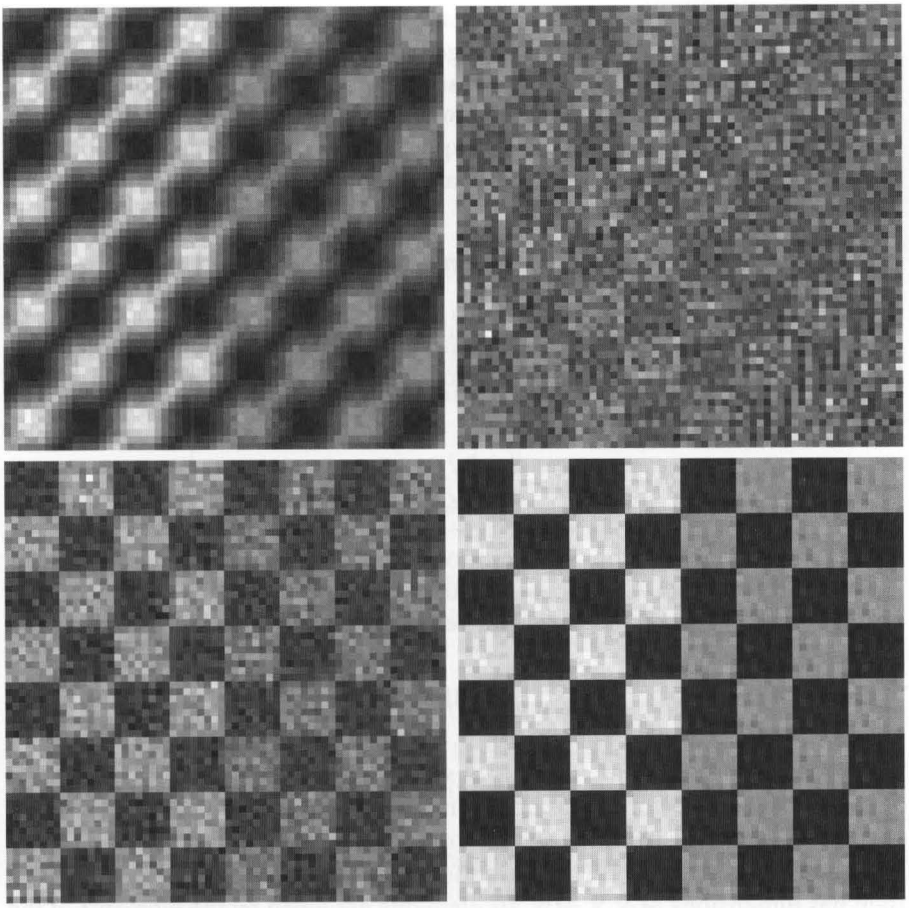
EXAMPLE 5.8: Using function deconvwnr to restore a blurred, noisy image.

■ Figure 5.8(a) was generated in the same way as Fig. 5.7(d), and Fig. 5.8(b) was obtained using the command

```
>> frest1 = deconvwnr(g, PSF);
```

where g is the corrupted image and PSF is the point spread function computed in Example 5.7. As noted earlier in this section, frest1 is the result of direct inverse filtering and, as expected, the result is dominated by the effects of noise. (As in Example 5.7, all displayed images were processed with pixeldup to zoom their size to 512 × 512 pixels.)

The ratio, R, discussed earlier in this section, was obtained using the original and noise images from Example 5.7:



a b
c d

FIGURE 5.8
(a) Blurred, noisy image. (b) Result of inverse filtering. (c) Result of Wiener filtering using a constant ratio. (d) Result of Wiener filtering using autocorrelation functions.

```
>> Sn = abs(fft2(noise)).^2;          % noise power spectrum
>> nA = sum(Sn(:))/numel(noise);      % noise average power
>> Sf = abs(fft2(f)).^2;              % image power spectrum
>> fA = sum(Sf(:))/numel(f);          % image average power.
>> R = nA/fA;
```

To restore the image using this ratio we write

```
>> frest2 = deconvwnr(g, PSF, R);
```

As Fig. 5.8(c) shows, this approach gives a significant improvement over direct inverse filtering.

Finally, we use the autocorrelation functions in the restoration (note the use of `fftshift` for centering):

```
>> NCORR = fftshift(real(ifft2(Sn)));
>> ICORR = fftshift(real(ifft2(Sf)));
>> frest3 = deconvwnr(g, PSF, NCORR, ICORR);
```

As Fig. 5.8(d) shows, the result is much closer to the original, but some noise is still evident. Because the original image and noise functions were known, we were able to estimate the correct parameters, and Fig. 5.8(d) is the best that can be accomplished with Wiener deconvolution in this case. The challenge in practice, when one (or more) of these quantities is not known, is the choice of functions used in experimenting, until an acceptable result is obtained. ■

5.8 Constrained Least Squares (Regularized) Filtering

Another well-established approach to linear restoration is *constrained least squares filtering*, called *regularized filtering* in toolbox documentation. We know from Section 3.4.1 that the 2-D discrete convolution of two functions f and h is

Recall that convolution is commutative, so the order of f and h does not matter.

$$h(x, y) \star f(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n)h(x - m, y - n)$$

where "$\star$" denotes the convolution operation. Using this equation, we can express the linear degradation model discussed in Section 5.1, $g(x, y) = h(x, y) \star f(x, y) + \eta(x, y)$, in vector-matrix form, as

$$\mathbf{g} = \mathbf{H}\mathbf{f} + \boldsymbol{\eta}$$

For example, suppose that $f(x, y)$ is of size $M \times N$. Then we can form the first N elements of the vector $\mathbf{f}$ by using the image elements in the first row of $f(x, y)$, the next N elements from the second row, and so on. The resulting vector will have dimensions $MN \times 1$. These are the dimensions of $\mathbf{g}$ and $\boldsymbol{\eta}$ also. Matrix $\mathbf{H}$ then has dimensions $MN \times MN$. Its elements are given by the elements of the preceding convolution equation.

It would be reasonable to conclude that the restoration problem can be reduced to simple matrix manipulations. Unfortunately, this is not the case. For instance, suppose that we are working with images of medium size; say $M = N = 512$. Then the vectors would be of dimension $262{,}144 \times 1$ and matrix $\mathbf{H}$ would be of dimensions $262{,}144 \times 262{,}144$. Manipulating vectors and matrices of these sizes is not a trivial task. The problem is complicated further by the fact that the inverse of $\mathbf{H}$ does not always exist due to zeros in the transfer function (see Section 5.6). However, formulating the restoration problem in matrix form does facilitate derivation of restoration techniques.

Although we do not derive the method of constrained least squares that we are about to present, central to this method is the issue of the sensitivity of the inverse of $\mathbf{H}$ mentioned in the previous paragraph. One way to deal with this issue is to base optimality of restoration on a measure of smoothness, such as the second derivative of an image (e.g., the Laplacian). To be meaningful, the restoration must be constrained by the parameters of the problems at hand.

Thus, what is desired is to find the minimum of a criterion function, C, defined as

$$C = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\nabla^2 f(x, y)\right]^2$$

subject to the constraint

$$\|\mathbf{g} - \mathbf{H}\hat{\mathbf{f}}\|^2 = \|\boldsymbol{\eta}\|^2$$

where $\|\mathbf{w}\|^2 = \mathbf{w}^T\mathbf{w}$ is the Euclidean vector norm,[†] $\hat{\mathbf{f}}$ is the estimate of the undegraded image, and the Laplacian operator ∇^2 is as defined in Section 3.5.1.

The frequency domain solution to this optimization problem is given by the expression

$$\hat{F}(u, v) = \left[\frac{H^*(u, v)}{|H(u, v)|^2 + \gamma|P(u, v)|^2}\right] G(u, v)$$

where γ is a parameter that must be adjusted so that the constraint is satisfied (if γ is zero we have an inverse filter solution), and $P(u, v)$ is the Fourier transform of the function

$$p(x, y) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

We recognize this function as the Laplacian operator introduced in Section 3.5.1. The only unknowns in the preceding formulation are γ and $\|\boldsymbol{\eta}\|^2$. However, it can be shown that γ can be found iteratively if $\|\boldsymbol{\eta}\|^2$, which is proportional to the noise power (a scalar), is known.

Constrained least squares filtering is implemented in the toolbox by function `deconvreg`, which has the syntax

```
frest = deconvreg(g, PSF, NOISEPOWER, RANGE)
```



where `g` is the corrupted image, `frest` is the restored image, `NOISEPOWER` is proportional to $\|\boldsymbol{\eta}\|^2$, and `RANGE` is the range of values where the algorithm is limited to look for a solution for γ. The default range is $[10^{-9}, 10^{9}]$ (`[1e-9, 1e9]` in MATLAB notation). If the last two parameters are excluded from the argument, `deconvreg` produces an inverse filter solution. A good starting estimate for `NOISEPOWER` is $MN[\sigma_\eta^2 + m_\eta^2]$ where M and N are the dimensions of the image and the parameters inside the brackets are the noise variance and noise squared mean. This estimate is a starting value and, as the following example shows, the final value used can be quite different.

[†] For a column vector $\mathbf{w}$ with n components, $\mathbf{w}^T\mathbf{w} = \sum_{k=1}^{n} w_k^2$, where w_k is the kth component of $\mathbf{w}$.

EXAMPLE 5.9: Using function `deconvreg` to restore a blurred, noisy image.

■ We now restore the image in Fig. 5.7(d) using `deconvreg`. The image is of size 64×64 and we know from Example 5.7 that the noise has a variance of 0.001 and zero mean. So, our initial estimate of `NOISEPOWER` is $(64)^2(0.001 + 0) \approx 4$. Figure 5.9(a) shows the result of using the command

```
>> frest1 = deconvreg(g, PSF, 4);
```

where `g` and `PSF` are from Example 5.7. The image was improved somewhat from the original, but obviously this is not a particularly good value for `NOISEPOWER`. After some experimenting with this parameter and parameter `RANGE`, we arrived at the result in Fig. 5.9(b), which was obtained using the command

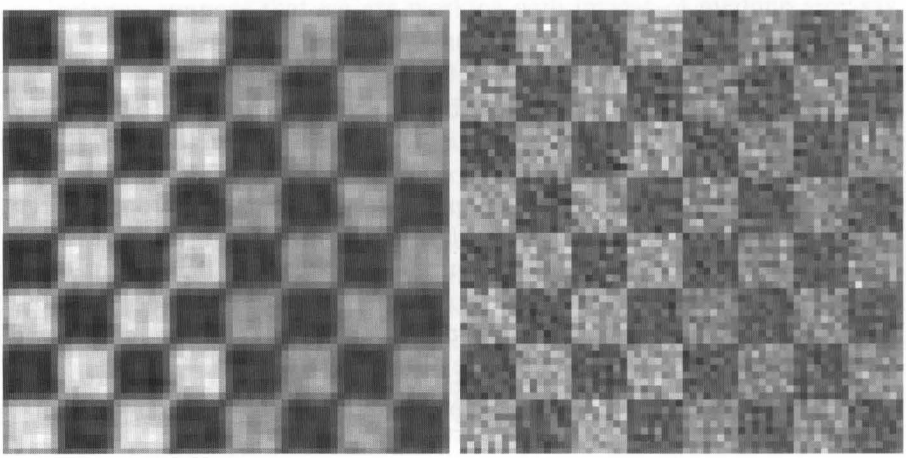
```
>> frest2 = deconvreg(g, PSF, 0.4, [1e-7 1e7]);
```

Thus we see that we had to go down one order of magnitude on `NOISEPOWER`, and `RANGE` was tighter than the default. The Wiener filtering result in Fig. 5.8(d) is superior, but we obtained that result with full knowledge of the noise and image spectra. Without that information, the results obtainable by experimenting with the two filters often are comparable [see Fig. 5.8(c)]. ■

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it helps sometimes to use function `edgetaper` (see Section 5.7) prior to calling `deconvreg`.

5.9 Iterative Nonlinear Restoration Using the Lucy-Richardson Algorithm

The image restoration methods discussed in the previous three sections are linear. They also are "direct" in the sense that, once the restoration filter is specified, the solution is obtained via one application of the filter. This simplicity of implementation, coupled with modest computational requirements and a



a b

FIGURE 5.9 (a) The image in Fig. 5.7(d) restored using a regularized filter with `NOISEPOWER` equal to 4. (b) The same image restored with a `NOISEPOWER` equal to 0.4 and a `RANGE` of `[1e–7 1e7]`.

well-established theoretical base, have made linear techniques a fundamental tool in image restoration for many years.

Nonlinear iterative techniques have been gaining acceptance as restoration tools that often yield results superior to those obtained with linear methods. The principal objections to nonlinear methods are that their behavior is not always predictable, and that they generally require significant computational resources. The first objection often loses importance based on the fact that nonlinear methods have been shown to be superior to linear techniques in a broad spectrum of applications (Jansson [1997]). The second objection has become less of an issue due to the dramatic increase in inexpensive computing available today. The nonlinear method of choice in the toolbox is a technique developed by Richardson [1972] and by Lucy [1974], working independently. The toolbox refers to this method as the Lucy-Richardson (L-R) algorithm, but you will see it also quoted in the literature as the Richardson-Lucy algorithm.

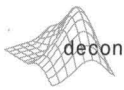
The L-R algorithm arises from a maximum-likelihood formulation (see Section 5.10) in which the image is modeled with Poisson statistics. Maximizing the likelihood function of the model yields an equation that is satisfied when the following iteration converges:

$$\hat{f}_{k+1}(x,y) = \hat{f}_k(x,y)\left[h(-x,-y) \star \frac{g(x,y)}{h(x,y) \star \hat{f}_k(x,y)}\right]$$

As before, "★" indicates convolution, $\hat{f}$ is the estimate of the undegraded image, and both g and h are as defined in Section 5.1. The iterative nature of the algorithm is evident. Its nonlinear nature arises from the division by $h \star \hat{f}$ on the right side of the equation.

As with most nonlinear methods, the question of when to stop the L-R algorithm is difficult to answer in general. One approach is to observe the output and stop the algorithm when a result acceptable in a given application has been obtained.

The L-R algorithm is implemented in the toolbox by function `deconvlucy`, which has the basic syntax




```
f = deconvlucy(g, PSF, NUMIT, DAMPAR, WEIGHT)
```

where `f` is the restored image, `g` is the degraded image, `PSF` is the point spread function, `NUMIT` is the number of iterations (the default is 10), and `DAMPAR` and `WEIGHT` are defined as follows.

`DAMPAR` is a scalar that specifies the threshold deviation of the resulting image from image `g`. Iterations are suppressed for the pixels that deviate within the `DAMPAR` value from their original value. This suppresses noise generation in such pixels, preserving image detail. The default is 0 (no damping).

`WEIGHT` is an array of the same size as `g` that assigns a weight to each pixel to reflect its quality. For example, a bad pixel resulting from a defective imaging array can be excluded from the solution by assigning to it a zero weight value. Another useful application of this array is to let it adjust the weights

of the pixels according to the amount of flat-field correction that may be necessary based on knowledge of the imaging array. When simulating blurring with a specified PSF (see Example 5.7), `WEIGHT` can be used to eliminate from computation pixels that are on the border of an image and thus are blurred differently by the PSF. If the PSF is of size $n \times n$ the border of zeros used in `WEIGHT` is of width `ceil(n/2)`. The default is a unit array of the same size as input image `g`.

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it helps sometimes to use function `edgetaper` (see Section 5.7) prior to calling `deconvlucy`.

EXAMPLE 5.10: Using function `deconvlucy` to restore a blurred, noisy image.

■ Figure 5.10(a) shows an image generated using the command

```
>> g = checkerboard(8);
```

which produced a square image of size 64×64 pixels. As before, the size of the image was increased to size 512×512 for display purposes by using function `pixeldup`:

```
>> imshow(pixeldup(g, 8))
```

The following command generated a Gaussian PSF of size 7×7 with a standard deviation of 10:

```
>> PSF = fspecial('gaussian', 7, 10);
```

Next, we blurred image `g` using `PDF` and added to it Gaussian noise of zero mean and standard deviation of 0.01:

```
>> SD = 0.01;
>> g = imnoise(imfilter(g, PSF), 'gaussian', 0, SD^2);
```

Figure 5.10(b) shows the result.

The remainder of this example deals with restoring image `g` using function `deconvlucy`. For `DAMPAR` we specified a value equal to 10 times `SD`:

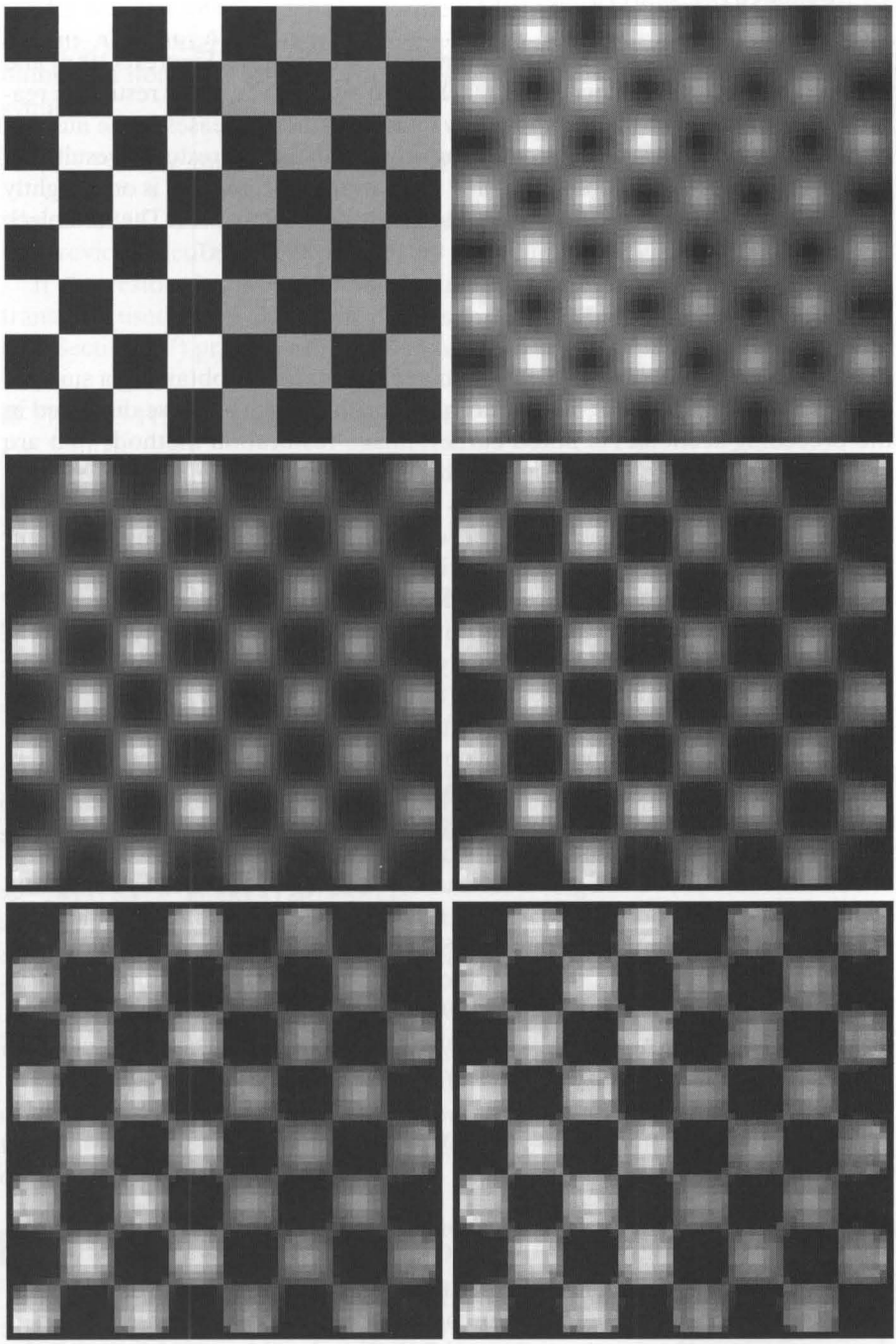
```
>> DAMPAR = 10*SD;
```

Array `WEIGHT` was created using the approach discussed in the preceding explanation of this parameter:

```
>> LIM = ceil(size(PSF, 1)/2);
>> WEIGHT = zeros(size(g));
>> WEIGHT(LIM + 1:end - LIM, LIM + 1:end - LIM) = 1;
```

Array `WEIGHT` is of size 64×64 with a border of 0s 4 pixels wide; the rest of the pixels are 1s.

The only variable left is `NUMIT`, the number of iterations. Figure 5.10(c) shows the result obtained using the commands



a b
c d
e f

FIGURE 5.10 (a) Original image. (b) Image blurred and corrupted by Gaussian noise. (c) through (f) Image (b) restored using the L-R algorithm with 5, 10, 20, and 100 iterations, respectively.

```
>> NUMIT = 5;
>> f5 = deconvlucy(g, PSF, NUMIT, DAMPAR, WEIGHT);
>> imshow(pixeldup(f5, 8), [])
```

Although the image has improved somewhat, it is still blurry. Figures 5.10(d) and (e) show the results obtained using `NUMIT = 10` and `20`. The latter result is a reasonable restoration of the blurred, noisy image. Further increases in the number of iterations produced more modest improvements in the restored result. For example, Fig. 5.10(f) was obtained using 100 iterations. This image is only slightly sharper and brighter than the result obtained using 20 iterations. The thin black border seen in all results was caused by the 0s in array `WEIGHT`. ■

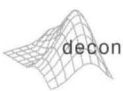
5.10 Blind Deconvolution

One of the most difficult problems in image restoration is obtaining a suitable estimate of the PSF to use in restoration algorithms such as those discussed in the preceding sections. As noted earlier, image restoration methods that are not based on specific knowledge of the PSF are called *blind deconvolution* algorithms.

A fundamental approach to blind deconvolution is based on maximum-likelihood estimation (MLE), an optimization strategy used for obtaining estimates of quantities corrupted by random noise. Briefly, an interpretation of MLE is to think of image data as random quantities having a certain likelihood of being produced from a family of other possible random quantities. The likelihood function is expressed in terms of $g(x, y)$, $f(x, y)$, and $h(x, y)$ (see Section 5.1), and the problem then is to find the maximum of the likelihood function. In blind deconvolution, the optimization problem is solved iteratively with specified constraints and, assuming convergence, the specific $f(x, y)$ and $h(x, y)$ that result in a maximum are the restored image *and* the PSF.

A derivation of MLE blind deconvolution is outside the scope of the present discussion, but you can gain a solid understanding of this area by consulting the following references: For background on maximum-likelihood estimation, see the classic book by Van Trees [1968]. For a review of some of the original image-processing work in this area see Dempster et al. [1977], and for some of its later extensions see Holmes [1992]. A good general reference book on deconvolution is Jansson [1997]. For detailed examples on the use of deconvolution in microscopy and in astronomy, see Holmes et al. [1995] and Hanisch et al. [1997], respectively.

The toolbox performs blind deconvolution using function `deconvblind`, which has the basic syntax




```
[f, PSF] = deconvblind(g, INITPSF)
```

where `g` is the degraded image, `INITPSF` is an initial estimate of the point spread function, `PSF` is the final computed estimate of this function, and `f` is the image restored using the estimated PSF. The algorithm used to obtain the restored image is the L-R iterative restoration algorithm explained in Section

5.9. The PSF estimation is affected strongly by the size of its initial guess, and less by its values (an array of 1s is a reasonable starting guess).

The number of iterations performed with the preceding syntax is 10 by default. Additional parameters may be included in the function to control the number of iterations and other features of the restoration, as in the following syntax:

```
[f, PSF] = deconvblind(g, INITPSF, NUMIT, DAMPAR, WEIGHT)
```

where `NUMIT`, `DAMPAR`, and `WEIGHT` are as described for the L-R algorithm in the previous section.

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it helps sometimes to use function `edgetaper` (see Section 5.7) prior to calling `deconvblind`.

EXAMPLE 5.11: Using function `deconvblind` to estimate a PSF.

■ Figure 5.11(a) is the PSF used to generate the degraded image shown in Fig. 5.10(b):

```
>> PSF = fspecial('gaussian', 7, 10);
>> imshow(pixeldup(PSF, 73), [ ])
```

As in Example 5.10, the degraded image in question was obtained with the commands

```
>> SD = 0.01;
>> g = imnoise(imfilter(g, PSF), 'gaussian', 0, SD^2);
```

In the present example we are interested in using function `deconvblind` to obtain an estimate of the PSF, given only the degraded image `g`. Figure 5.11(b) shows the PSF resulting from the following commands:

```
>> INITPSF = ones(size(PSF));
>> NUMIT = 5;
>> [g5, PSF5] = deconvblind(g, INITPSF, NUMIT, DAMPAR, WEIGHT);
>> imshow(pixeldup(PSF5, 73), [ ])
```

where we used the same values as in Example 5.10 for `DAMPAR` and `WEIGHT`.

Figures 5.11(c) and (d), displayed in the same manner as `PSF5`, show the PSF obtained with 10, and 20 iterations, respectively. The latter result is close to the true PSF in Fig. 5.11(a) (it is easier to compare the images by looking at their corners, rather than their centers). ■

5.11 Image Reconstruction from Projections

Thus far in this chapter we have dealt with the problem of image restoration. In this section interest switches to the problem of *reconstructing* an image from a series of 1-D projections. This problem, typically called *computed tomography* (CT), is one of the principal applications of image processing in medicine.

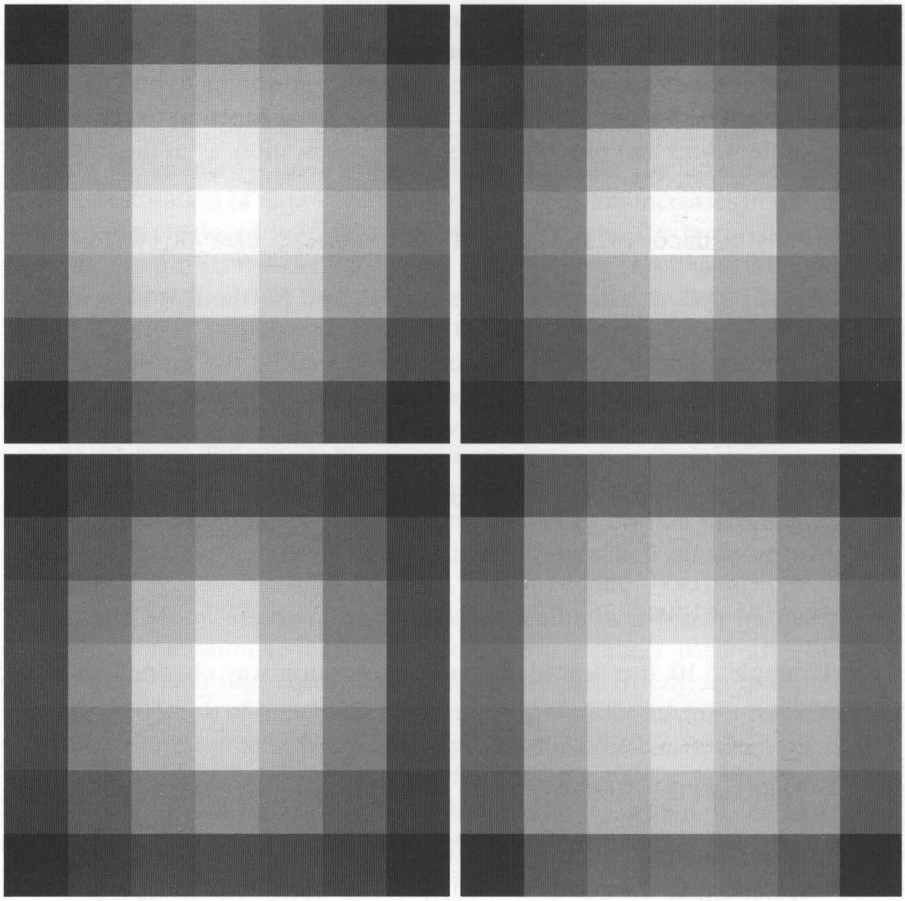
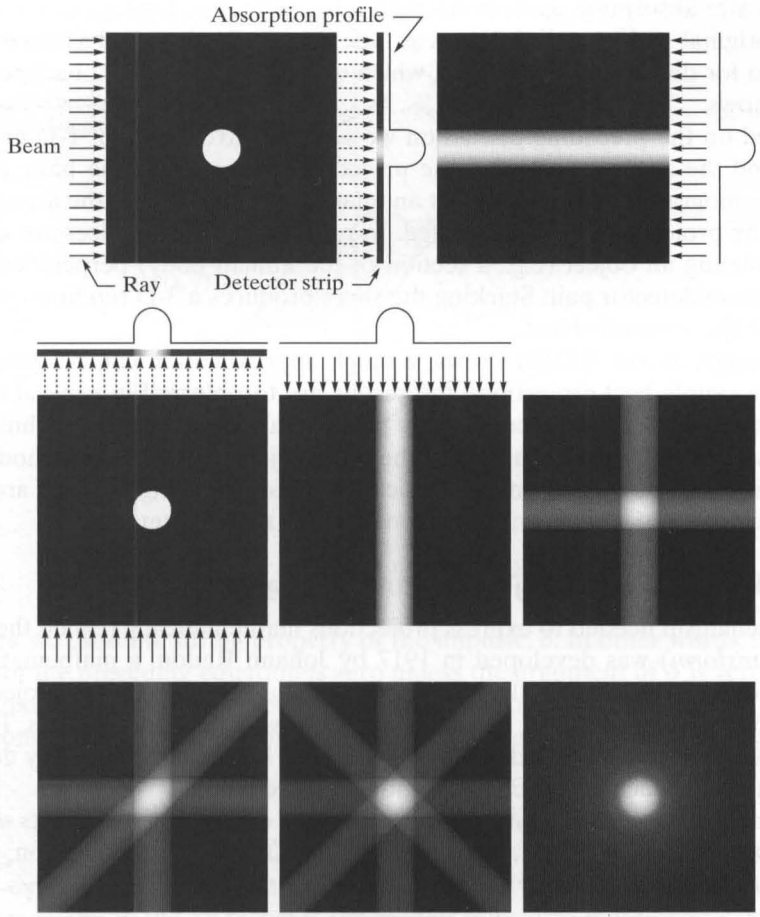


FIGURE 5.11
(a) Original PSF. (b) through (d) Estimates of the PSF using 5, 10, and 20 iterations in function `deconvblind`.

5.11.1 Background

The foundation of image reconstruction from projections is straightforward and can be explained intuitively. Consider the region in Fig. 5.12(a). To give physical meaning to the following discussion, assume that this region is a "slice" through a section of a human body showing a tumor (bright area) embedded in a homogeneous area of tissue (black background). Such a region might be obtained, for example, by passing a thin, flat beam of X-rays perpendicular to the body, and recording at the opposite end measurements proportional to the absorption of the beam as it passes through the body. The tumor absorbs more of the X-ray energy, hence giving a higher reading for absorption, as the signal (*absorption profile*) on the right side of Fig. 5.12(a) shows. Observe that maximum absorption occurs through the center of the region, where the beam encounters the longest path through the tumor. At this point, the absorption profile is all the information we have about the object.



a b
c d e
f g h

FIGURE 5.12
(a) Flat region with object, parallel beam, detector strip, and absorption profile. (b) Backprojection of absorption profile. (c) Beam and detector strip rotated 90° and (d) Backprojection of absorption profile. (e) Sum of (b) and (d). (f) Result of adding another backprojection (at 45°). (g) Result of adding yet another backprojection at 135°. (h) Result of adding 32 backprojections 5.625° apart.

There is no way of determining from a single projection whether we are dealing with a single object or multiple objects along the path of the beam, but we start the reconstruction based on this partial information. The approach is to create an *image* by projecting the absorption profile back along the direction of the original beam, as Fig. 5.12(b) shows. This process, called *backprojection*, generates a 2-D digital image from a 1-D absorption profile waveform. By itself, this image is of little value. However, suppose that we rotate the beam/detector arrangement by 90° [Fig. 5.12(c)] and repeat the backprojection process. By adding the resulting backprojection to Fig. 5.12(b) we obtain the image in Fig. 5.12(e). Note how the intensity of the region containing the object is twice the intensity of the other major components of the image.

It is intuitive that we should be able to refine the preceding results by generating more backprojections at different angles. As Figs. 5.12(f)-(h) show, this is precisely what happens. As the number of backprojections increases, the area

with greater absorption gains in strength relatively to the homogeneous areas in the original region until those areas fade into the background as the image is scaled for display, as Fig. 5.12(h), which was obtained using 32 backprojections, shows.

Based on the preceding discussion we see that, given a set of 1-D projections, and the angles at which those projections were taken, the basic problem in tomography is to reconstruct an image (called a *slice*) of the area from which the projections were generated. In practice, numerous slices are taken by translating an object (e.g., a section of the human body) perpendicularly to the beam/detector pair. Stacking the slices produces a 3-D rendition of the inside of the scanned object.

Although, as Fig. 5.12(h) shows, a rough approximation can be obtained by using simple backprojections, the results are too blurred in general to be of practical use. Thus, the tomography problem also encompasses techniques for reducing the blurring inherent in the backprojection process. Methods for describing backprojections mathematically and for reducing blurring are the principal topics of discussion in the remainder of this chapter.

5.11.2 Parallel-Beam Projections and the Radon Transform

The mechanism needed to express projections mathematically (called the *Radon Transform*) was developed in 1917 by Johann Radon, a mathematician from Vienna, who derived the basic mathematical expressions for projecting a 2-D object along parallel rays as part of his work on line integrals. These concepts were "rediscovered" over four decades later during the early development of CT machines in England and the United States.

A straight line in Cartesian coordinates can be described either by its *slope-intercept* form, $y = ax + b$, or, as in Fig. 5.13, by its *normal* representation,

$$x\cos\theta + y\sin\theta = \rho$$

The projection of a parallel-ray beam can be modeled by a set of such lines, as Fig. 5.14 shows. An arbitrary *point* in the projection profile at coordinates (ρ_j, θ_k) is given by the *ray sum* along the line $x\cos\theta_k + y\sin\theta_k = \rho_j$. The ray sum is a line integral, given by

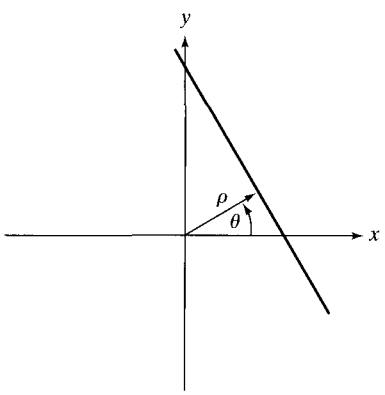


FIGURE 5.13 Normal representation of a straight line.

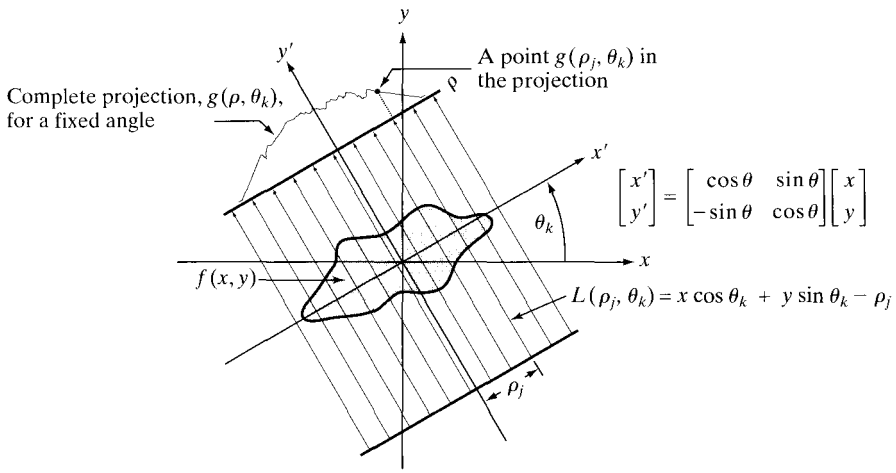


FIGURE 5.14 Geometry of a parallel-ray beam and its corresponding projection.

In this section we follow CT convention and place the origin in the center of an image, instead of our customary top, left. Because both are right-handed coordinate systems, we can account for their difference via a translation of the origin.

$$g(\rho_j, \theta_k) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y)\,\delta(x\cos\theta_k + y\sin\theta_k - \rho_j)\,dxdy$$

where we used the sifting property of the impulse, δ. In other words, the right side of the preceding equation is zero unless the argument of δ is zero, meaning that the integral is computed only along the line $x\cos\theta_k + y\sin\theta_k = \rho_j$. If we consider all values of ρ and θ the preceding equation generalizes to

$$g(\rho, \theta) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y)\,\delta(x\cos\theta + y\sin\theta - \rho)\,dxdy$$

This expression, which gives the projection (line integral) of $f(x, y)$ along an arbitrary line in the xy-plane, is the *Radon transform* mentioned earlier. As Fig. 5.14 shows, the complete projection for an arbitrary angle, θ_k, is $g(\rho, \theta_k)$, and this function is obtained by inserting θ_k in the Radon transform.

A discrete approximation to the preceding integral may be written as:

$$g(\rho, \theta) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x, y)\,\delta(x\cos\theta + y\sin\theta - \rho)$$

where x, y, ρ, and θ are now discrete variables. Although this expression is not useful in practice,[†] it does provide a simple model that we can use to explain how projections are generated. If we fix θ and allow ρ to vary, we see that this expression yields the sum of all values of $f(x, y)$ along the line defined

[†]When dealing with discrete images, the variables are integers. Thus, the argument of the impulse will seldom be zero, and the projections would not to be along a line. Another way of saying this is that the discrete formulation shown does not provide an adequate representation of projections along a line in discrete space. Numerous formulations exist to overcome this problem, but the toolbox function that computes the Radon transform (called `radon` and discussed in Section 5.11.6) takes the approach of approximating the continuous Radon transform and using its linearity properties to obtain the Radon transform of a digital image as the sum of the Radon transform of its individual pixels. The reference page of function `radon` gives an explanation of the procedure.

by the values of these two parameters. Incrementing through all values of ρ required to span the region defined by $f(x, y)$ (with θ fixed) yields *one* projection. Changing θ and repeating the procedure yields another projection, and so on. Conceptually, this approach is how the projections in Fig. 5.12 were generated.

Returning to our explanation, keep in mind that the objective of tomography is to recover $f(x, y)$ from a given set of projections. We do this by creating an image from each 1-D projection by backprojecting that particular projection [see Figs. 5.12(a) and (b)]. The images are then summed to yield the final result, as we illustrated in Fig. 5.12. To obtain an expression for the back-projected image, let us begin with a *single* point, $g(\rho_j, \theta_k)$, of the *complete* projection, $g(\rho, \theta_k)$, for a fixed value of θ_k (see Fig. 5.14). Forming part of an image by backprojecting this single point is nothing more than copying the line $L(\rho_j, \theta_k)$ onto the image, where the value of *all* points along the line is $g(\rho_j, \theta_k)$. Repeating this process for all values of ρ_j in the projected signal (while keeping the value of θ fixed at θ_k) result in the following expression:

$$\begin{aligned} f_{\theta_k}(x, y) &= g(\rho, \theta_k) \\ &= g(x \cos\theta_k + y \sin\theta_k, \theta_k) \end{aligned}$$

for the image resulting from backprojecting the projection just discussed. This equation holds for an arbitrary value of θ_k, so we can write in general that the image formed from a *single* backprojection (obtained at angle θ) is given by

$$f_\theta(x, y) = g(x \cos\theta + y \sin\theta, \theta)$$

We obtain the final image by integrating over all the back-projected images:

$$f(x, y) = \int_0^\pi f_\theta(x, y)\, d\theta$$

where the integral is taken only over half a revolution because the projections obtained in the intervals $[0, \pi]$ and $[\pi, 2\pi]$ are identical.

In the discrete case, the integral becomes a sum of all the back-projected images:

$$f(x, y) = \sum_{\theta=0}^{\pi} f_\theta(x, y)$$

This is a summation of entire images and, therefore, does not have the problems explained in the preceding footnote in connection with our simple, discrete approximation of the continuous Radon transform.

where the variables are now discrete. Because the projections at 0° and 180° are mirror images of each other, the summation is carried out to the last angle increment before 180°. For example, if 0.5° angle increments are used, the summation is from 0° to 179.5° in half-degree increments. Function `radon` (see Section 5.11.6) and the preceding equation were used to generate the images in Fig. 5.12. As is evident in that figure, especially in Fig. 5.12(h), using this procedure yields unacceptably blurred results. Fortunately, as you will see in the following section, significant improvements are possible by reformulating the backprojection approach.

5.11.3 The Fourier Slice Theorem and Filtered Backprojections

The 1-D Fourier transform of $g(\rho, \theta)$ with respect to ρ is given by

$$G(\omega, \theta) = \int_{-\infty}^{\infty} g(\rho, \theta) e^{-j2\pi\omega\rho} \, d\rho$$

where ω is the frequency variable and it is understood that this expression is for a fixed value of θ.

A fundamental result in computed tomography, known as the *Fourier slice theorem*, states that Fourier transform of a projection [i.e., $G(\omega, \theta)$ in the preceding equation] is a *slice* of the 2-D transform of the region from which the projection was obtained [i.e., $f(x, y)$]; that is,

See Gonzalez and Woods [2008] for a derivation of the Fourier slice theorem.

$$\begin{aligned} G(\omega, \theta) &= \left[F(u, v)\right]_{u=\omega\cos\theta;\, v=\omega\sin\theta} \\ &= F(\omega\cos\theta,\, \omega\sin\theta) \end{aligned}$$

where, as usual, $F(u, v)$ is the 2-D Fourier transform of $f(x, y)$. Figure 5.15 illustrates this result graphically.

Next, we use the Fourier slice theorem to derive an expression for obtaining $f(x, y)$ in the frequency domain. Given $F(u, v)$ we can obtain $f(x, y)$ using the inverse Fourier transform:

$$f(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(u, v) e^{j2\pi(ux + vy)} \, du \, dv$$

If, as above, we let $u = \omega\cos\theta$ and $v = \omega\sin\theta$, then $du\,dv = \omega\, d\omega\, d\theta$ and we can express the preceding integral in polar coordinates as

The relationship $du\,dv = \omega\, d\omega\, d\theta$ is from integral calculus, where the Jacobian is used as the basis for a change of variables.

$$f(x, y) = \int_{0}^{2\pi} \int_{0}^{\infty} F(\omega\cos\theta, \omega\sin\theta) e^{j2\pi\omega(x\cos\theta + y\sin\theta)} \omega \, d\omega \, d\theta$$

Then, from the Fourier slice theorem,

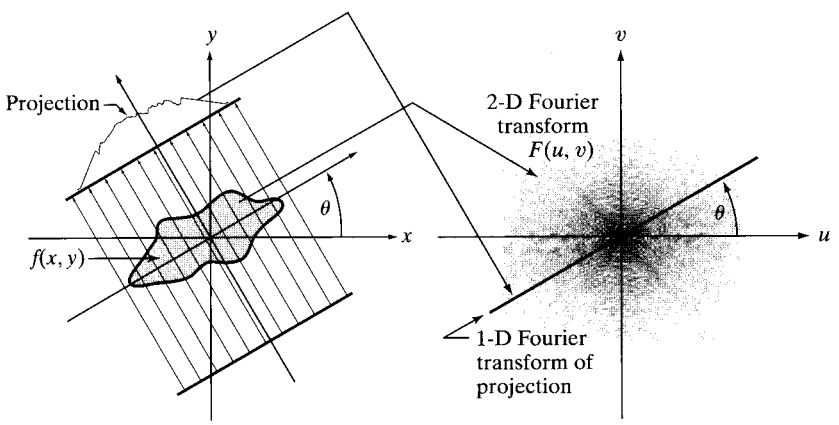


FIGURE 5.15 Graphical illustration of the Fourier slice theorem.

$$f(x,y) = \int_0^{2\pi} \int_0^{\infty} G(\omega,\theta)\, e^{j2\pi\omega(x\cos\theta + y\sin\theta)}\, \omega\, d\omega\, d\theta$$

By splitting this integral into two expressions, one for θ in the range 0 to π and the other from π to 2π, and using the fact that $G(\omega, \theta + \pi) = G(-\omega, \theta)$, we can express the preceding integral as

$$f(x,y) = \int_0^{\pi} \int_{-\infty}^{\infty} |\omega|\, G(\omega,\theta)\, e^{j2\pi\omega(x\cos\theta + y\sin\theta)} d\omega\, d\theta$$

In terms of integration with respect to ω, the term $x\cos\theta + y\sin\theta$ is a constant, which we also recognize as ρ. Therefore, we can express the preceding equation as

$$f(x,y) = \int_0^{\pi} \left[\int_{-\infty}^{\infty} |\omega|\, G(\omega,\theta)\, e^{j2\pi\omega\rho} d\omega \right]_{\rho = x\cos\theta + y\sin\theta} d\theta$$

The inner expression is in the form of a 1-D inverse Fourier transform, with the added term $|\omega|$ which, from the discussion in Chapter 4, we recognize as a 1-D *filter* function in the frequency domain. This function (which has the shape of a "V" extending infinitely in both directions) in not integrable. Theoretically, this problem is handled by using so-called *generalized delta functions*. In practice, we window the function so that it becomes zero outside a specified range. We address the filtering problem in the next section.

The preceding equation is a basic result in parallel-beam tomography. It states that $f(x, y)$, the complete back-projected image resulting from a set of parallel-beam projections, can be obtained as follows:

1. Compute the 1-D Fourier transform of each projection.
2. Multiply each Fourier transform by the filter function, $|\omega|$. This filter must be multiplied by a suitable windowing function, as explained in the next section.
3. Obtain the inverse 1-D Fourier transform of each filtered transform resulting from step 2.
4. Obtain $f(x, y)$ by integrating (summing) all the 1-D inverse transforms from step 3.

Because a filter is used, the method just presented is appropriately referred to as image reconstruction by *filtered projections*. In practice, we deal with discrete data, so all frequency domain computations are implemented using a 1-D FFT algorithm.

5.11.4 Filter Implementation

The filtering component of the filtered backprojection approach developed in the previous section is the foundation for dealing with the blurring problem discussed earlier, which is inherent in unfiltered backprojection reconstruction.

The shape of filter $|\omega|$ is a ramp, a function that is not integrable in the continuous case. In the discrete case, the function obviously is limited in length and its existence is not an issue. However, this filter has the undesirable characteristic that its amplitude increases linearly as a function of frequency, thus making it susceptible to noise. In addition, limiting the width of the ramp implies that it is multiplied by a box window in the frequency domain, which we know has undesirable ringing properties in the spatial domain. As noted in the previous section, the approach taken in practice is to multiply the ramp filter by a windowing function that *tapers* the "tails" of the filter, thus reducing its amplitude at high frequencies. This helps from the standpoint of both noise and ringing. The toolbox supports sinc, cosine, Hamming, and Hann windows. The duration (width) of the ramp filter itself is limited by the number of frequency points used to generate the filter.

The sinc window has the transfer function

$$H_S(\omega) = \frac{\sin\left(\pi\omega/2\Delta\omega K\right)}{\left(\pi\omega/2\Delta\omega K\right)}$$

for $\omega = 0, \pm\Delta\omega, \pm 2\Delta\omega, \ldots, \pm K\Delta\omega$, where K is the number of frequency intervals (the number of points minus one) in the filter. Similarly, the cosine window is given by

$$H_C(\omega) = \cos\frac{\pi\omega}{2\Delta\omega K}$$

The Hamming and Hann windows have the same basic equation:

$$H(\omega) = c + (c-1)\cos\frac{2\pi\omega}{\Delta\omega K}$$

When $c = 0.54$ the window is called a *Hamming window*; when $c = 0.5$, the window is called a *Hann* window. The difference between them is that in the Hann window the end points are 0, whereas the Hamming window has a small offset. Generally, results using these two windows are visually indistinguishable.

Figure 5.16 shows the backprojection filters generated by multiplying the preceding windowing functions by the ramp filter. It is common terminology to refer to the ramp filter as the *Ram-Lak filter*, after Ramachandran and Lakshminarayanan [1971], who generally are credited with having been first to suggest it. Similarly, a filter based on using the sinc window is called the Shepp-Logan filter, after Shepp and Logan [1974].

5.11.5 Reconstruction Using Fan-Beam Filtered Backprojections

The parallel-beam projection approach discussed in the previous sections was used in early CT machines and still is the standard for introducing concepts and developing the basic mathematics of CT reconstruction. Current CT systems are based on fan-beam geometries capable of yielding superior resolution, low signal-to-noise ratios, and fast scan times. Figure 5.17 shows a typical

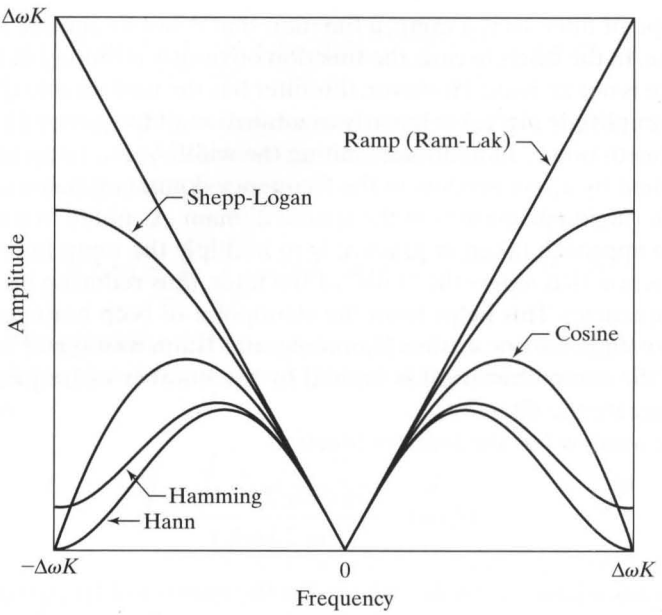


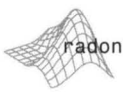
FIGURE 5.16 Various filters used for filtered backprojections. The filters shown were obtained by multiplying the Ramp filter by the various windowing functions discussed in the preceding paragraphs.

fan-beam scanning geometry that employs a ring of detectors (typically on the order of 5000 individual detectors). In this arrangement, the X-ray source rotates around the patient. For each horizontal increment of displacement a full revolution of the source generates a slice image. Moving the patient perpendicularly to the plane of the detectors generates a set of slice images that, when stacked, yield a 3-D representation of the scanned section of the body.

Derivation of the equations similar to the ones developed in the previous sections for parallel beams is not difficult, but the schematics needed to explain the process are tedious. Detailed derivations can be found in Gonzalez and Woods [2008] and in Prince and Links [2006]. An important aspect of these derivations is that they establish a one-to-one correspondence between the fan-beam and parallel geometries. Going from one to the other involves a simple change of variables. As you will learn in the following section, the toolbox supports both geometries.

5.11.6 Function `radon`

Function `radon` is used to generate a set of parallel-ray projections for a given 2-D rectangular array (see Fig. 5.14). The basic syntax for this function is




```
R = radon(I, theta)
```

where `I` is a 2-D array and `theta` is a 1-D array of angle values. The projections are contained in the columns of `R`, with the number of projections generated being equal to the number of angles in array `theta`. The projections generated

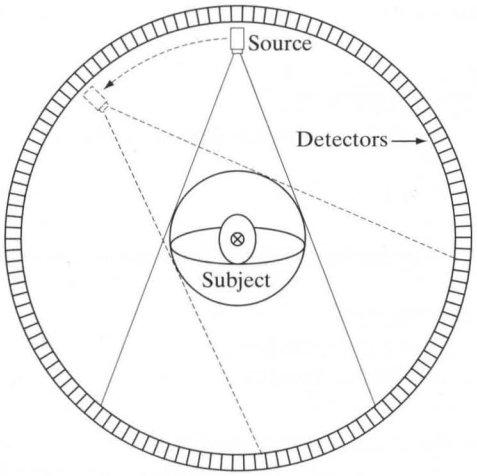


FIGURE 5.17
A typical CT geometry based on fan-beam projections.

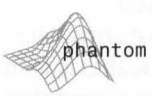
are long enough to span the widest view seen as the beam is rotated. This view occurs when the rays are perpendicular to the main diagonal of the array rectangle. In other words, for an input array of size $M \times N$, the minimum length that the projections can have is $[M^2 + N^2]^{1/2}$. Of course, projections at other angles in reality are shorter, and those are padded with 0s so that all projections are of the same length (as required for `R` to be a rectangular array). The actual length returned by function `radon` is slightly larger than the length of the main diagonal to account for the unit area of each pixel.

Function `radon` also has a more general syntax:

```
[R, xp] = radon(I, theta)
```

where `xp` contains the values of the coordinates along the x'-axis, which are the values of ρ in Fig. 5.14. As example 5.12 below shows, the values in `xp` are useful for labeling plot axes.

A useful function for generating a well-known image (called a *Shepp-Logan head phantom*) used in CT algorithm simulations has the syntax




```
P = phantom(def, n)
```

where `def` is a string that specifies the type of head phantom generated, and `n` is the number of rows and columns (the default is 256). Valid values of string `def` are

- `'Shepp-Logan'`—Test image used widely by researchers in tomography. The contrast in this image is very low.
- `'Modified Shepp-Logan'`—Variant of the Shepp-Logan phantom in which the contrast is improved for better visual perception.

EXAMPLE 5.12: Using function `radon`.

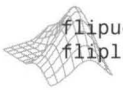
■ The following two images are shown in Figs. 5.18(a) and (c).

```
>> g1 = zeros(600, 600);
>> g1(100:500, 250:350) = 1;
>> g2 = phantom('Modified Shepp-Logan', 600);
>> imshow(g1)
>> figure, imshow(g2)
```

The Radon transforms using half-degree increments are obtained as follows:

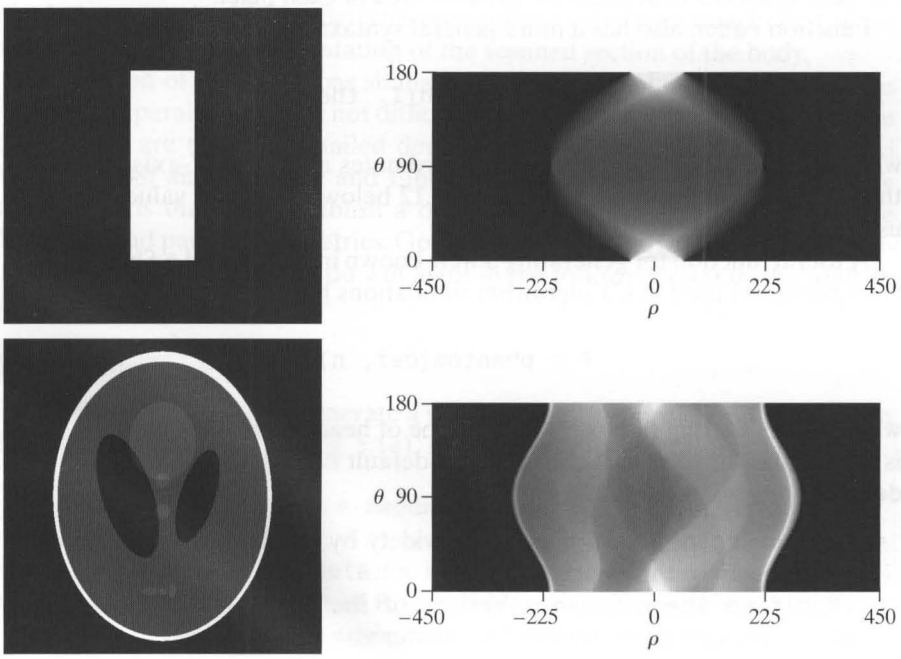
```
>> theta = 0:0.5:179.5;
>> [R1, xp1] = radon(g1, theta);
>> [R2, xp2] = radon(g2, theta);
```

`B = flipud(A)` returns A with the rows flipped about the horizontal axis. `B = fliplr(A)` returns A with the columns flipped about the vertical axis.



The first column of `R1` is the projection for $\theta = 0°$, the second column is the projection for $\theta = 0.5°$, and so on. The first element of the first column corresponds to the most negative value of ρ and the last is its largest positive value, and similarly for the other columns. If we want to display `R1` so that the projections run from left to right, as in Fig. 5.14, and the first projection appears in the bottom of the image, we have to transpose and flip the array, as follows:

```
>> R1 = flipud(R1');
>> R2 = flipud(R2');
>> figure, imshow(R1, [],'XData', xp1([1 end]),'YData', [179.5 0])
```



a b
c d

FIGURE 5.18 Illustration of function `radon`. (a) and (c) Two images; (b) and (d) their corresponding Radon transforms. The vertical axis is in degrees and the horizontal axis is in pixels.

```
>> axis xy
>> axis on
>> xlabel('\rho'), ylabel('\theta')
>> figure, imshow(R2, [],'XData', xp2([1 end]),'YData', [179.5 0])
>> axis xy
>> axis on
>> xlabel('\rho'), ylabel('\theta')
```

Function `axis xy` moves the origin of the axis system to the bottom right from its top, left default location. See the comments on this function in Example 3.4.

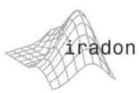
Figures 5.18(b) and (d) show the results. Keeping in mind that each row in these two images represents a complete projection for a fixed value of θ, observe, for example, how the widest projection in Fig. 5.18(b) occurs when $\theta = 90°$, which corresponds to the parallel beam intersecting the broad side of the rectangle. Radon transforms displayed as images of the form in Figs. 5.18(b) and (c) often are called *sinograms*. ■

5.11.7 Function `iradon`

Function `iradon` reconstructs an image (slice) from a given set of projections taken at different angles; in other words, `iradon` computes the inverse Radon transform. This function uses the filtered backprojection approach discussed in Sections 5.11.3 and 5.11.4. The filter is designed directly in the frequency domain and then multiplied by the FFT of the projections. All projections are zero-padded to a power of 2 before filtering to reduce spatial domain aliasing and to speed up FFT computations.

The basic `iradon` syntax is

```
I = iradon(R, theta, interp, filter, frequency_scaling, output_size)
```



where the parameters are as follows:

- `R` is the backprojection data, in which the columns are 1-D backprojections organized as a function of increasing angle from left to right.
- `theta` describes the angles (in degrees) at which the projections were taken. It can be either a vector containing the angles or a scalar specifying `D_theta`, the incremental angle between projections. If `theta` is a vector, it must contain angles with equal spacing between them. If `theta` is a scalar specifying `D_theta`, it is assumed that the projections were taken at angles `theta = m*D_theta`, where `m = 0,1,2,...,size(R, 2) - 1`. If the input is the empty matrix (`[]`), `D_theta` defaults to `180/size(R, 2)`.
- `interp` is a string that defines the interpolation method used to generate the final reconstructed image. The principal values of `interp` are listed in Table 5.4.
- `filter` specifies the filter used in the filtered-backprojection computation. The filters supported are those summarized in Fig. 5.16, and the strings used to specify them in function `iradon` are listed in Table 5.5. If option `'none'` is specified, reconstruction is performed without filtering. Using the syntax

See Section 6.6 regarding interpolation.

TABLE 5.4
Interpolation methods used in function `iradon`.

Method	Description
`'nearest'`	Nearest-neighbor interpolation.
`'linear'`	Linear interpolation (this is the default).
`'cubic'`	Cubic interpolation.
`'spline'`	Spline interpolation.

TABLE 5.5
Filters supported by function `iradon`.

Name	Description
`'Ram-Lak'`	This is the ramp filter discussed in Section 5.11.4, whose frequency response is $\|w\|$. This is the default filter.
`'Shepp-Logan'`	Filter generated by multiplying the Ram-Lak filter by a sinc function.
`'Cosine'`	Filter generated by multiplying the Ram-Lak filter by a cosine function.
`'Hamming'`	Filter generated by multiplying the Ram-Lak filter by a Hamming window.
`'Hann'`	Filter generated by multiplying the Ram-Lak filter by a Hann window.
`'None'`	No filtering is performed.

```
[I, H] = iradon(...)
```

returns the frequency response of the filter in vector `H`. We used this syntax to generate the filter responses in Fig. 5.16.

Frequency scaling is used to lower the cutoff frequency of the reconstruction filter for the purpose of reducing noise in the projections. Frequency scaling makes the ideal ramp response more of a lowpass filter, achieving noise reduction at the expense of spatial resolution along the ρ-axis.

- `frequency_scaling` is a scalar in the range `(0,1]` that modifies the filter by rescaling its frequency axis. The default is `1`. If `frequency_scaling` is less than `1`, the filter is compressed to fit into the frequency range `[0, frequency_scaling]`, in normalized frequencies; all frequencies above `frequency_scaling` are set to `0`.
- `output_size` is a scalar that specifies the number of rows and columns in the reconstructed image. If `output_size` is not specified, the size is determined from the length of the projections:

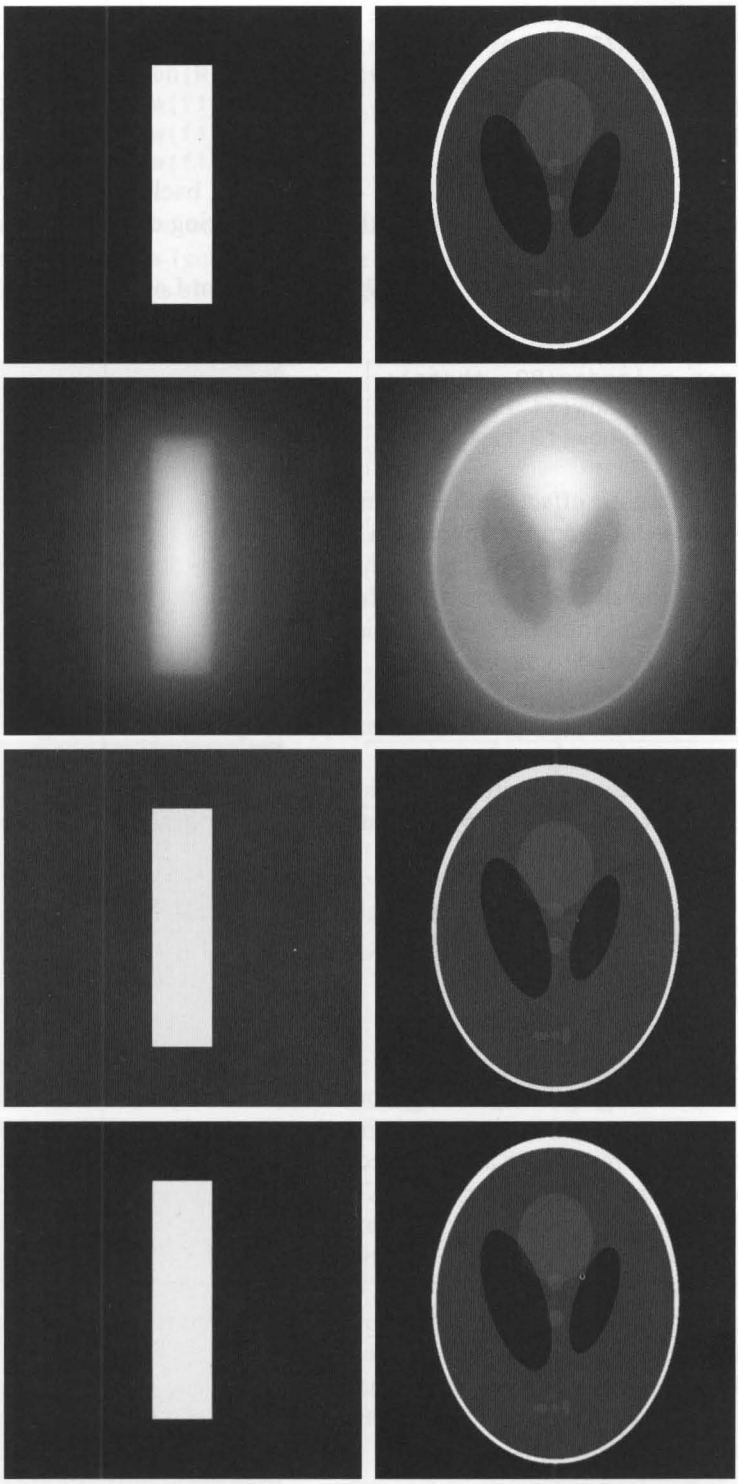
```
output_size = 2*floor(size(R,1)/(2*sqrt(2)))
```

If you specify `output_size`, `iradon` reconstructs a smaller or larger portion of the image but does not change the scaling of the data. If the projections were calculated with the `radon` function, the reconstructed image may not be the same size as the original image.

EXAMPLE 5.13: Using function `iradon`.

■ Figures 5.19(a) and (b) show the two images from Fig. 5.18. Figures 5.19(c) and (d) show the results of the following sequence of steps:

```
>> theta = 0:0.5:179.5;
>> R1 = radon(g1, theta);
```



a b
c d
e f
g h

FIGURE 5.19
The advantages of filtering. (a) Rectangle, and (b) Phantom images. (c) and (d) Backprojection images obtained without filtering. (e) and (f) Backprojection images obtained using the default (Ram-Lak) filter. (g) and (h) Results obtained using the Hamming filter option.

```
>> R2 = radon(g2, theta);
>> f1 = iradon(R1, theta, 'none');
>> f2 = iradon(R2, theta, 'none');
>> figure, imshow(f1, [])
>> figure, imshow(f2, [])
```

These two figures illustrate the effects of computing backprojections without filtering. As you can see, they exhibit the same blurring characteristics as the images in Fig. 5.12.

Adding even the crudest of filters (the default Ram-Lak filter),

```
>> f1_ram = iradon(R1, theta);
>> f2_ram = iradon(R2, theta);
>> figure, imshow(f1_ram, [])
>> figure, imshow(f2_ram, [])
```

can have a dramatic effect on the reconstruction results, as Figs. 5.19(e) and (f) show. As expected from the discussion at the beginning of Section 5.11.4, the Ram-Lak filter produces ringing, which you can see as faint ripples, especially in the center top and bottom regions around the rectangle in Fig. 5.19(e). Note also that the background in this figure is lighter than in all the others. The reason can be attributed to display scaling, which moves the average value up as a result of significant negative values in the ripples just mentioned. This grayish tonality is similar to what you encountered in Chapter 3 with scaling the intensities of Laplacian images.

The situation can be improved considerably by using any of the other filters in Table 5.5. For example, Figs. 5.19(g) and (h) were generated using a Hamming filter:

```
>> f1_hamm = iradon(R1, theta, 'Hamming');
>> f2_hamm = iradon(R2, theta, 'Hamming');
>> figure, imshow(f1_hamm, [])
>> figure, imshow(f2_hamm, [])
```

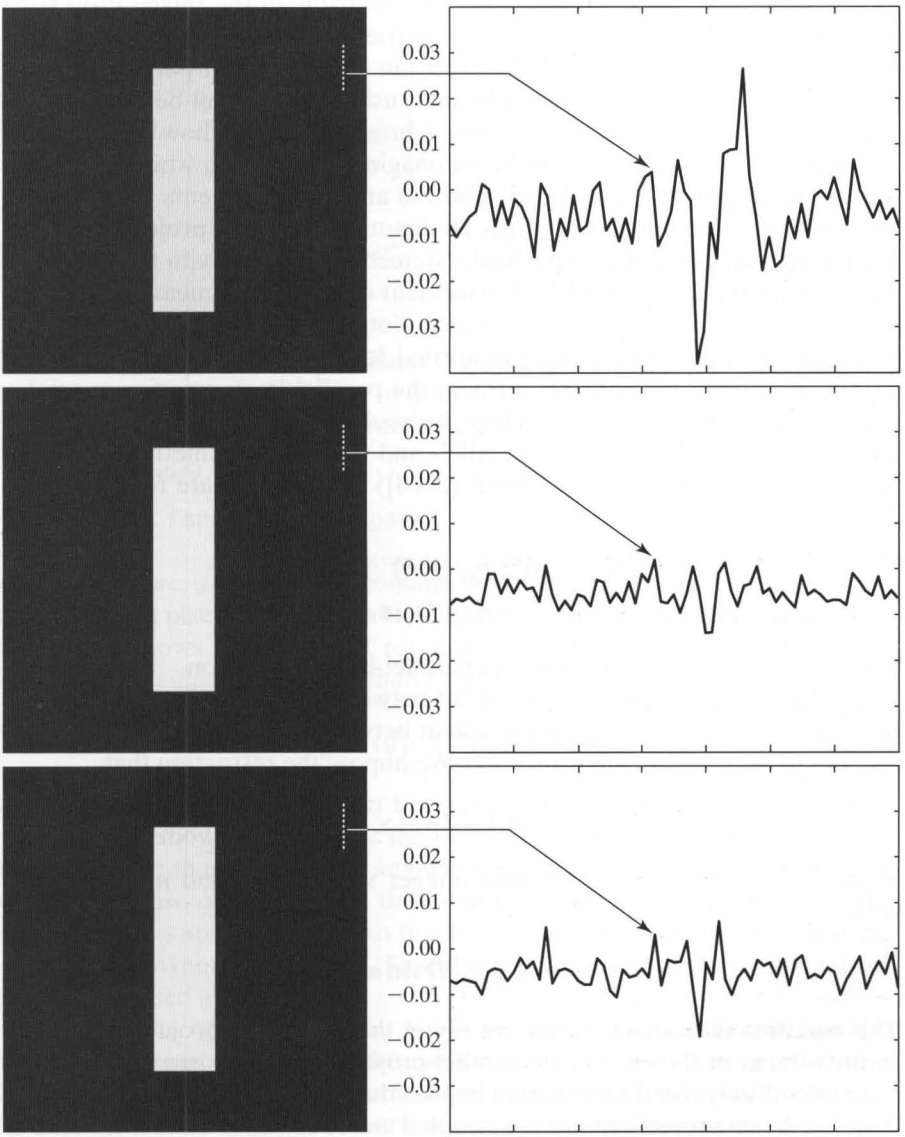
The results in these two figures are a significant improvement. There still is slightly visible ringing in Fig. 5.19(g), but it is not as objectionable. The phantom image does not show as much ringing because its intensity transitions are not as sharp and rectilinear as in the rectangle.

Interpolation is used by `iradon` as part of backprojection computations. Recall from Fig. 5.14 that projections are onto the ρ-axis, so the computation of backprojections starts with the points on those projections. However, values of a projection are available only at set of a discrete locations along the ρ-axis. Thus, interpolating the data along the ρ-axis is required for proper assignment of values to the pixels in the back-projected image.

To illustrate the effects of interpolation, consider the reconstruction of `R1` and `R2` (generated earlier in this example) using the first three interpolation methods in Table 5.4:

```
>> f1_near = iradon(R1, theta,'nearest');
>> f1_lin = iradon(R1, theta,'linear');
>> f1_cub = iradon(R1, theta,'cubic');
>> figure, imshow(f1_near,[])
>> figure, imshow(f1_lin,[])
>> figure, imshow(f1_cub,[])
```

The results are shown on the left column of Fig. 5.20. The plots on the right are intensity profiles (generated using function `improfile`) along the short vertical line segments shown in the figures on the left. Keeping in mind that



a b
c d
e f

FIGURE 5.20 Left column: Backprojected images obtained with function `iradon` using the default (Ram-Lak) filter and three methods of interpolation: (a) nearest neighbor, (c) linear, and (e) cubic. Right column: Intensity profiles along the vertical segments shown dotted in the images on the left. Ringing is quite apparent in the center section of the profile in (b).

the background of the original image is constant, we see that linear and cubic interpolation produced better results than nearest neighbor interpolation, in the sense that the former two methods yielded intensity variations in the background that are smaller (i.e., closer to constant) than those produced by the latter method. The default (linear) interpolation often produces results that are visually indistinguishable from those of cubic and spline interpolation, and linear interpolation runs significantly faster. ■

5.11.8 Working with Fan-Beam Data

The geometry of a fan-beam imaging system was introduced in Section 5.11.5. In this section we discuss briefly the tools available in the Image Processing Toolbox for working with fan-bean geometries. Given fan-beam data, the approach used by the toolbox is to convert fan beams to their parallel counterparts. Then, backprojections are obtained using the parallel-beam approach discussed earlier. In this section we give a brief overview of how this is done.

Figure 5.21 shows a basic fan-beam imaging geometry in which the detectors are arranged on a circular arc and the angular increments of the source are assumed to be equal. Let $p_{\text{fan}}(\alpha, \beta)$ denote a fan-beam projection, where α is the angular position of a particular detector measured with respect to the *center ray*, and β is the angular displacement of the source, measured with respect to the y-axis, as shown in the figure. Note that a ray in the fan beam can be represented as a line, $L(\rho, \theta)$, in normal form (see Fig. 5.13), which is the approach we used to represent a ray in the parallel-beam imaging geometry discussed in Section 5.11.2. Therefore, it should not be a surprise that there is a correspondence between the parallel- and fan-beam geometries. In fact, it can be shown (Gonzalez and Woods [2008]) that the two are related by the expression

$$\begin{aligned} p_{\text{fan}}(\alpha, \beta) &= p_{\text{par}}(\rho, \theta) \\ &= p_{\text{par}}(D \sin \alpha, \alpha + \beta) \end{aligned}$$

where $p_{\text{par}}(\rho, \theta)$ is the corresponding parallel-beam projection.

Let $\Delta\beta$ denote the angular increment between successive fan-beam projections and let $\Delta\alpha$ be the angular increment between rays, which determines the number of samples in each projection. We impose the restriction that

$$\Delta\beta = \Delta\alpha = \gamma$$

Then, $\beta = m\gamma$ and $\alpha = n\gamma$ for some integer values of m and n, and we can write

$$p_{\text{fan}}(n\gamma, m\gamma) = p_{\text{par}}\left[D \sin n\gamma, (m + n)\gamma\right]$$

This equation indicates that the nth ray in the mth radial projection is equal to the nth ray in the $(m + n)$th parallel projection. The $D \sin n\gamma$ term on the right side of the preceding equation implies that parallel projections converted from fan-beam projections are not sampled uniformly, an issue that can lead to

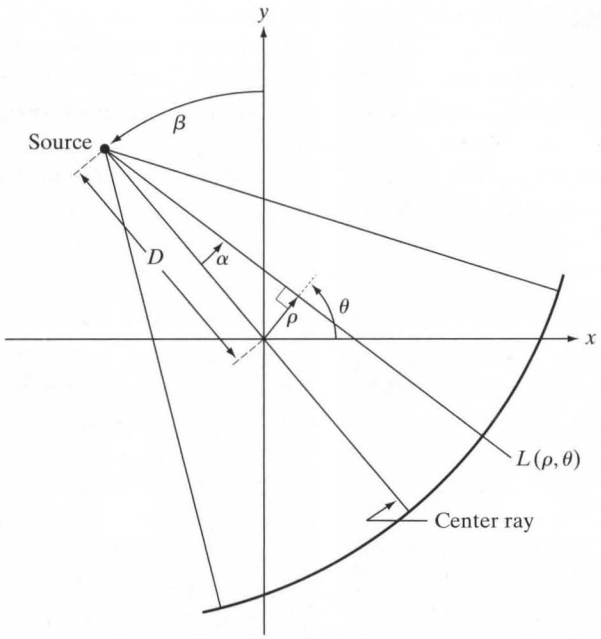
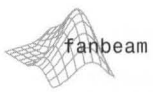


FIGURE 5.21
Details of a fan-beam projection arrangement.

blurring, ringing, and aliasing artifacts if the sampling intervals $\Delta\alpha$ and $\Delta\beta$ are too coarse, as Example 5.15 later in this section illustrates.

Toolbox function `fanbeam` generates fan-beam projections using the following syntax:




```
B = fanbeam(g, D, param1,val1,param2,val2,...)
```

where, as before, `g` is the image containing the object to be projected, and `D` is the distance in pixels from the vertex of the fan beam to the center of rotation, as Fig. 5.22 shows. The center of rotation is assumed to be the center of the image. `D` is specified to be larger than half the diameter of `g`:

```
D = K*sqrt(size(g, 1)^2 + size(g,2)^2)/2
```

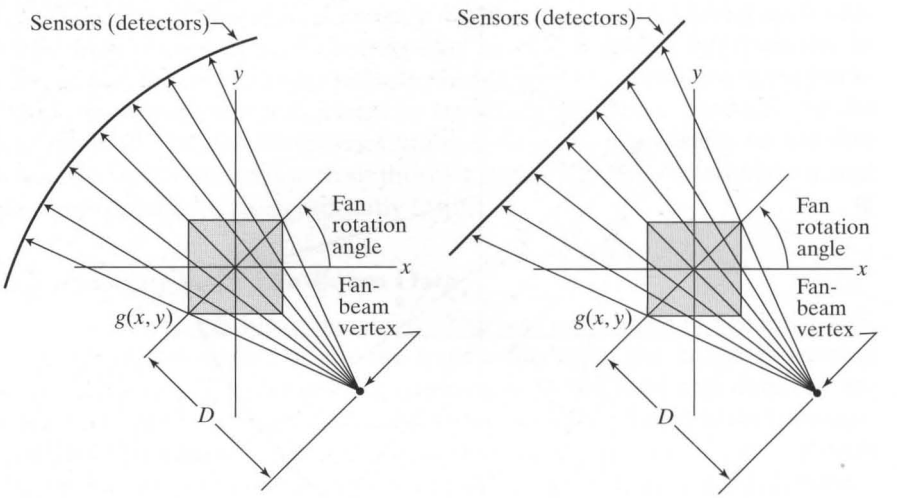
where `K` is a constant greater than `1` (e.g., `K = 1.5` to `2` are reasonable values).

Figure 5.22 shows the two basic fan-beam geometries supported by function `fanbeam`. Note that the rotation angle is specified counterclockwise from the x-axis (the sense of this angle is the same as the rotation angle in Fig. 5.21). The parameters and values for this function are listed in Table 5.6. Parameters `'FanRotationIncrement'` and `'FanSensorSpacing'` are the increment $\Delta\beta$ and $\Delta\alpha$ discussed above.

Each column of `B` contains the fan-beam sensor samples at one rotation angle. The number of columns in `B` is determined by the fan rotation increment. In the default case, `B` has 360 columns. The number of rows in `B` is determined by the number of sensors. Function `fanbeam` determines the number of sensors

a b

FIGURE 5.22
The arc and linear fan-beam projection capabilities of function `fanbeam`. $g(x, y)$ refers to the image region shown in gray.



by calculating how many beams are required to cover the entire image for any rotation angle. As you will see in the following example, this number depends strongly on the geometry (`line` or `arc`) specified.

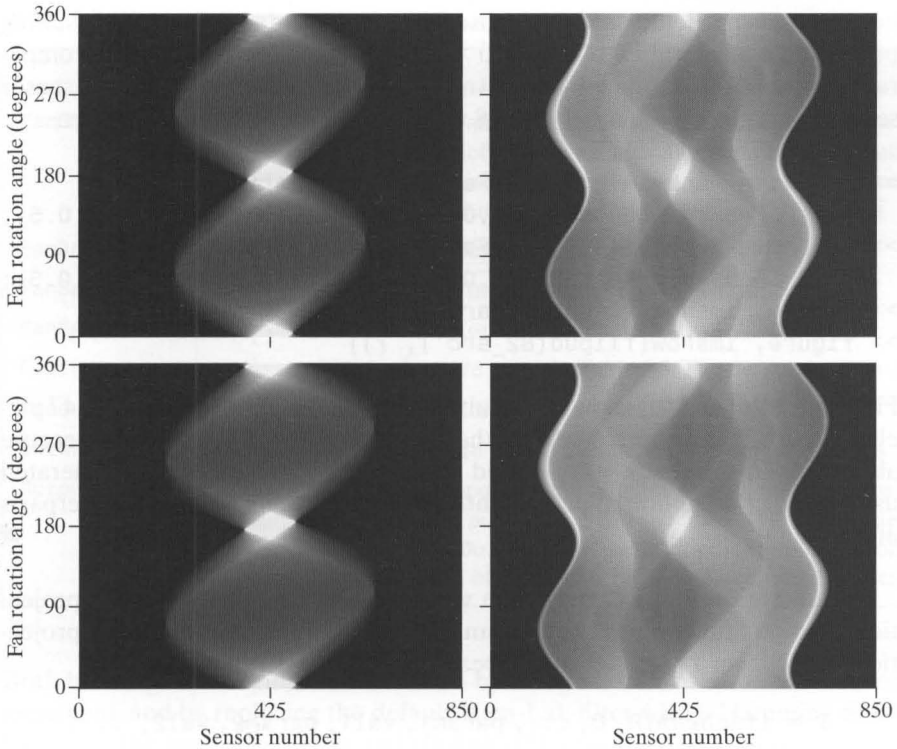
EXAMPLE 5.14: Working with function `fanbeam`.

■ Figures 5.23(a) and (b) were generated using the following commands:

```
>> g1 = zeros(600, 600);
>> g1(100:500, 250:350) = 1;
>> g2 = phantom('Modified Shepp-Logan',600);
>> D = 1.5*hypot(size(g1, 1), size(g1,2))/2;
>> B1_line = fanbeam(g1, D, 'FanSensorGeometry','line',...
          'FanSensorSpacing', 1, 'FanRotationIncrement', 0.5);
>> B1_line = flipud(B1_line');
>> B2_line = fanbeam(g2, D, 'FanSensorGeometry','line',...
```

TABLE 5.6
Parameters and values used in function `fanbeam`.

Parameter	Description and Values
`'FanRotationIncrement'`	Specifies the rotation angle increments of the fan-beam projections measured in degrees. Valid values are positive real scalars. The default value is 1.
`'FanSensorGeometry'`	A text string specifying how the equally-spaced sensors are arranged. Valid values are `'arc'` (the default) and `'line'`.
`'FanSensorSpacing'`	A positive real scalar specifying the spacing of the fan-beam sensors. If `'arc'` is specified for the geometry, then the value is interpreted as angular spacing in degrees. If `'line'` is specified, then the value is interpreted as linear spacing. The default in both cases is 1.



a b
c d

FIGURE 5.23 Illustration of function `fanbeam`. (a) and (b) Linear fan-beam projections for the rectangle and phantom images generated with function `fanbeam`. (c) and (d) Corresponding arc projections.

```
        'FanSensorSpacing', 1, 'FanRotationIncrement', 0.5);
>> B2_line = flipud(B2_line');
>> imshow(B1_line, [])
>> figure, imshow(B2_line, [])
```

See Example 5.12 for an explanation of why we transpose the image and use function `flipud`.

where `g1` and `g2` are the rectangle and phantom images in Figs. 5.18(a) and (c). As the preceding code shows, `B1` and `B2` are the fan-beam projections of the rectangle, generated using the `'line'` option, sensor spacing 1 unit apart (the default), and angle increments of 0.5°, which corresponds to the increments used to generate the parallel-beam projections (Radon transforms) in Figs. 5.18(b) and (d). Comparing these parallel-beam projections and the fan-beam projections in Figs. 5.23(a) and (b), we note several significant differences. First, the fan-beam projections cover a 360° span, which is twice the span shown for the parallel-beam projections; thus, the fan beam projections repeat themselves one time. More interestingly, note that the corresponding shapes are quite different, with the fan-beam projections appearing "skewed." This is a direct result of the fan- versus the parallel-beam geometries.

If you have difficulties visualizing why the fan-beam projections look as they do, the following exercise will help: (1) draw a set of fan-beam rays on a sheet of paper; (2) cut a small piece of paper in the form of the rectangle in Fig. 5.18(a); (3) place the rectangle in the center of the beams; and (4) rotate the rectangle in small increments, starting at 0°. Studying how the beams intersect the rectangles will clarify why the shapes of the fan-beam projections appear skewed.

As mentioned earlier, function `fanbeam` determines the number of sensors by calculating how many beams are required to cover the entire image for any rotation angle. The sizes of the images in Figs. 5.23(a) and (b) are 720×855 pixels. If, to generate beam projections using the `'arc'` option, we use the same separation

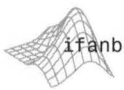
between sensor elements that we used for the `'line'` options, the resulting projection arrays will be of size 720×67. To generate arrays of sizes comparable to those obtained with the `'line'` option, we need to specify the sensor separation to be on the order of 0.08 units. The commands are as follows:

```
>> B1_arc = fanbeam(g1, D, 'FanSensorGeometry','arc',...
        'FanSensorSpacing', .08, 'FanRotationIncrement', 0.5);
>> B2_arc = fanbeam(g2, D, 'FanSensorGeometry','arc',...
        'FanSensorSpacing', .08, 'FanRotationIncrement', 0.5);
>> figure, imshow(flipud(B1_arc'), [])
>> figure, imshow(flipud(B2_arc'), [])
```

We used the same approach as in Example 5.12 to superimpose the axes and scales on the images in Fig. 5.23.

Figures 5.23(c) and (d) show the results. These images are of size 720×847 pixels; they are slightly narrower than the images in Figs. 5.23(a) and (b). Because all images in the figure were scaled to the same size, the images generated using the `'arc'` option appear slightly wider than their `'line'` counterparts after scaling. ■

Just as we used function `iradon` when dealing with parallel-beam projections, toolbox function `ifanbeam` can be used to obtain a filtered backprojection image from a given set of fan-beam projections. The syntax is




```
I = ifanbeam(B, D, ..., param1, val1, param2, val2, ...)
```

where, as before, `B` is an array of fan-beam projections and `D` is the distance in pixels from the vertex of the fan beam to the center of rotation. The parameters and their range of valid values are listed in Table 5.7.

EXAMPLE 5.15: Working with function `ifanbeam`.

■ Figure 5.24(a) shows a filtered backprojection of the head phantom, generated using the default values for functions `fanbeam` and `ifanbeam`, as follows:

```
>> g = phantom('Modified Shepp-Logan', 600);
>> D = 1.5*hypot(size(g, 1), size(g, 2))/2;
>> B1 = fanbeam(g, D);
>> f1 = ifanbeam(B1, D);
>> figure, imshow(f1, [])
```

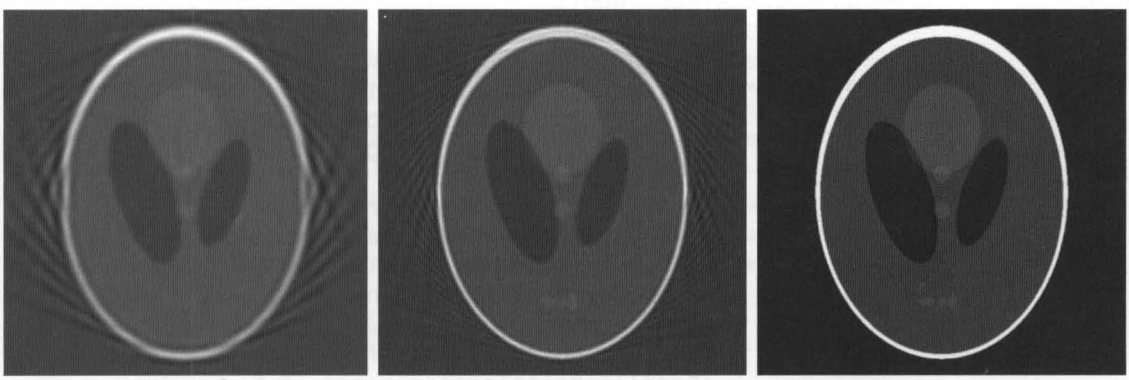
As you can see in Fig. 5.24(a), the default values were too coarse in this case to achieve a reasonable level of quality in the reconstructed image. Figure 5.24(b) was generated using the following commands:

```
>> B2 = fanbeam(g, D, 'FanRotationIncrement', 0.5,...
                  'FanSensorSpacing', 0.5);
>> f2 = ifanbeam(B2, D, 'FanRotationIncrement', 0.5,...
                'FanSensorSpacing', 0.5, 'Filter', 'Hamming');
>> figure, imshow(f2, [])
```

TABLE 5.7 Parameters and values used in function `ifanbeam`.

Parameter	Description and Values
`'FanCoverage'`	Specifies the range through which the beams are rotated. Valid values are `'cycle'` (the default) which indicates rotation in the full range $[0, 360°)$ and `'minimal'`, which indicates the minimum range necessary to represent the object from which the projections in `B` were generated.
`'FanRotationIncrement'`	As explained for function `fanbeam` in Table 5.6.
`'FanSensorGeometry'`	As explained for function `fanbeam` in Table 5.6.
`'FanSensorSpacing'`	As explained for function `fanbeam` in Table 5.6.
`'Filter'`	Valid values are given in Table 5.5. The default is `'Ram-Lak'`.
`'FrequencyScaling'`	As explained for function `iradon`.
`'Interpolation'`	Valid values are given in Table 5.4. The default value is `'linear'`.
`'OutputSize'`	A scalar that specifies the number of rows and columns in the reconstructed image. If `'OutputSize'` is not specified, `ifanbeam` determines the size automatically. If `'OutputSize'` is specified, `ifanbeam` reconstructs a smaller or larger portion of the image, but does not change the scaling of the data.

Both blurring and ringing were reduced by using smaller rotation and sensor increment, and by replacing the default Ram-Lak filter with a Hamming filter. However, the level of blurring and ringing still is unacceptable. Based on the results in Example 5.14, we know that the number of sensors specified when the `'arc'` option is used plays a significant role in the quality of the projections. In the following code we leave everything the same, with the exception of the separation between samples, which we decrease by a factor of ten:



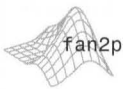
a b c

FIGURE 5.24 (a) Phantom image generated and reconstructed using the default values in functions `fanbeam` and `ifanbeam`. (b) Result obtained by specifying the rotation and sensor spacing increments at 0.5, and using a Hamming filter. (c) Result obtained with the same parameter values used in (b), except for the spacing between sensors, which was changed to 0.05.

```
>> B3 = fanbeam(g, D, 'FanRotationIncrement', 0.5,...
                                        'FanSensorSpacing', 0.05);
>> f3 = ifanbeam(B3, D, 'FanRotationIncrement', 0.5,...
                'FanSensorSpacing', 0.05, 'Filter', 'Hamming');
>> figure, imshow(f3, [])
```

As Fig. 5.24(c) shows, reducing the separation between sensors (i.e., *increasing* the number of sensors) resulted in an image of significantly improved quality. This is consistent with the conclusions in Example 5.14 regarding the importance of the number of sensors used in determining the effective "resolution" of the fan-beam projections. ■

Before concluding this section, we mention briefly two toolbox utility functions for converting between fan and parallel parallel projections. Function `fan2para` converts fan-beam data to parallel-beam data using the following syntax:




```
P = fan2para(F, D, param1, val1, param2, val2, ...)
```

where `F` is the array whose columns are fan-beam projections and `D` is the distance from the fan vertex to the center of rotation that was used to generate the fan projections, as discussed earlier in this section. Table 5.8 lists the parameters and corresponding values for this function.

EXAMPLE 5.16: Working with function `fan2para`.

■ We illustrate the use of function `fan2para` by converting the fan-beam projections in Figs. 5.23(a) and (d) into parallel-beam projections. We specify the parallel projection parameter values to correspond to the projections in Figs. 5.18(b) and (d):

```
>> g1 = zeros(600, 600);
>> g1(100:500, 250:350) = 1;
>> g2 = phantom('Modified Shepp-Logan',600);
>> D = 1.5*hypot(size(g1, 1), size(g1,2))/2;
>> B1_line = fanbeam(g1, D, 'FanSensorGeometry',...
             'line','FanSensorSpacing', 1, ...
             'FanRotationIncrement', 0.5);
>> B2_arc = fanbeam(g2, D, 'FanSensorGeometry', 'arc',...
             'FanSensorSpacing', .08, 'FanRotationIncrement',0.5);
>> P1_line = fan2para(B1_line, D, 'FanRotationIncrement', 0.5,...
             'FanSensorGeometry','line',...
             'FanSensorSpacing', 1,...
             'ParallelCoverage','halfcycle',...
             'ParallelRotationIncrement', 0.5,...
             'ParallelSensorSpacing',1);
>> P2_arc = fan2para(B2_arc, D, 'FanRotationIncrement', 0.5,...
             'FanSensorGeometry','arc',...
             'FanSensorSpacing', 0.08,...
             'ParallelCoverage','halfcycle',...
```

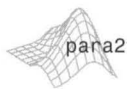
Parameter	Description and Values
`'FanCoverage'`	As explained for function `ifanbeam` in Table 5.7
`'FanRotationIncrement'`	As explained for function `fanbeam` in Table 5.6.
`'FanSensorGeometry'`	As explained for function `fanbeam` in Table 5.6.
`'FanSensorSpacing'`	As explained for function `fanbeam` in Table 5.6.
`'Interpolation'`	Valid values are given in Table 5.3. The default value is `'linear'`.
`'ParallelCoverage'`	Specifies the range of rotation: `'cycle'` means that the parallel data is to cover 360°, and `'halfcyle'` (the default), means that the parallel data covers 180°.
`'ParallelRotationIncrement'`	Positive real scalar specifying the parallel-beam rotation angle increment, measured in degrees. If this parameter is not included in the function argument, the increment is assumed to be the same as the increment of the fan-beam rotation angle.
`'ParallelSensorSpacing'`	A positive real scalar specifying the spacing of the parallel-beam sensors in pixels. If this parameter is not included in the function argument, the spacing is assumed to be uniform, as determined by sampling over the range implied by the fan angles.

TABLE 5.8 Parameters and values used in function `fan2para`.

```
            'ParallelRotationIncrement', 0.5,...
            'ParallelSensorSpacing',1);
>> P1_line = flipud(P1_line');
>> P2_arc = flipud(P2_arc');
>> figure, imshow(P1_line,[])
>> figure, imshow(P2_arc, [])
```

Note the use of function `flipud` to flip the transpose of the arrays, as we did in generating Fig. 5.18 so the data would correspond to the axes arrangement shown in that figure. Images `P1_line` and `P2_arc`, shown in Figs. 5.25(a) and (b), are the parallel-beam projections generated from the corresponding fan-beam projections `B1_line` and `B2_arc`. The dimensions of the images in Fig. 5.25 are the same as those in Fig. 5.18, so we do not show the axes and labels here. Note that the images are visually identical. ■

The procedure used to convert from a parallel-bean to a fan-beam is similar to the method just discussed. The function is




```
F = para2fan(P, D, param1, val1, param2, val2, ...)
```

where `P` is an array whose columns contain parallel projections and `D` is as before. Table 5.9 lists the parameters and allowed values for this function.

a
b

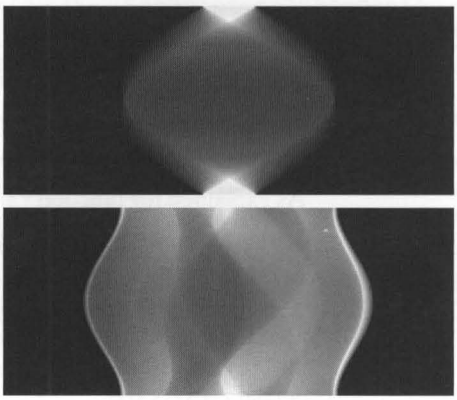


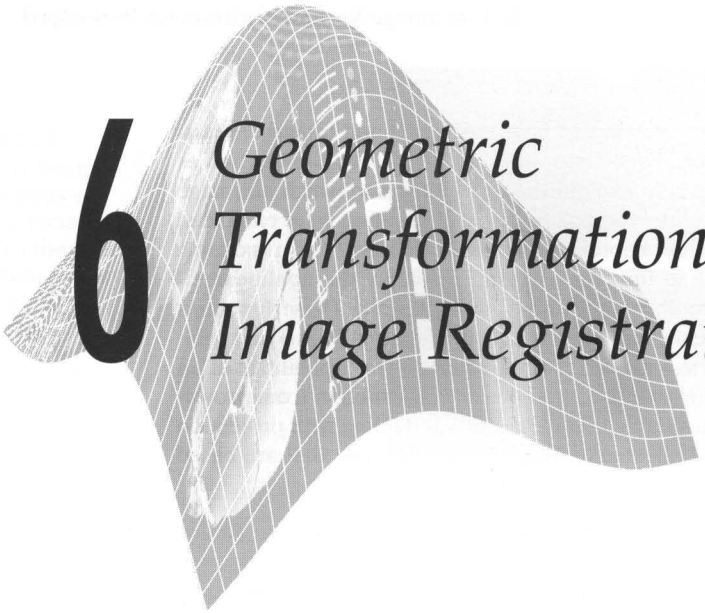
FIGURE 5.25
Parallel-beam projections of (a) the rectangle, and (b) the head phantom images, generated from the fan-beam projections in Figs. 5.23(a) and (d).

TABLE 5.9
Parameters and values used in function `para2fan`.

Parameter	Description and Values
`'FanCoverage'`	As explained for function `ifanbeam` in Table 5.7
`'FanRotationIncrement'`	Positive real scalar specifying the rotation angle increment of the fan-beam projections in degrees. If `'FanCoverage'` is 'cycle', then `'FanRotationIncrement'` must be a factor of 360. If this parameter is not specified, then it is set to the same spacing as the parallel-beam rotation angles.
`'FanSensorGeometry'`	As explained for function `fanbeam` in Table 5.6.
`'FanSensorSpacing'`	If the value is specified as `'arc'` or `'line'`, then the explanation for function `fanbeam` in Table 5.6 applies. If this parameter is not included in the function argument, the default is the smallest value implied by `'ParallelSensorSpacing'`, such that, if `'FanSensorGeometry'` is `'arc'`, then `'FanSensorSpacing'` is `180/PI*ASIN(PSPACE/D)` where `PSPACE` is the value of `'ParallelSensorSpacing'`. If `'FanSensorGeometry'` is `'line'`, then `'FanSensorSpacing'` is `D*ASIN(PSPACE/D)`.
`'Interpolation'`	Valid values are given in Table 5.4. The default value is `'linear'`.
`'ParallelCoverage'`	As explained for function `fan2para` in Table 5.8
`'ParallelRotationIncrement'`	As explained for function `fan2para` in Table 5.8.
`'ParallelSensorSpacing'`	As explained for function `fan2para` in Table 5.8.

Summary

The material in this chapter is a good overview of how MATLAB and Image Processing Toolbox functions can be used for image restoration, and how they can be used as the basis for generating models that help explain the degradation to which an image has been subjected. The capabilities of the toolbox for noise generation were enhanced significantly by the development in this chapter of functions `imnoise2` and `imnoise3`. Similarly, the spatial filters available in function `spfilt`, especially the nonlinear filters, are a significant extension of toolbox's capabilities in this area. These functions are perfect examples of how relatively simple it is to incorporate MATLAB and toolbox functions into new code to create applications that enhance the capabilities of an already large set of existing tools. Our treatment of image reconstruction from projections covers the principal functions available in the toolbox for dealing with projection data. The techniques discussed are applicable to modeling applications that are based on tomography.



6 Geometric Transformations and Image Registration

Preview

Geometric transformations modify the spatial relationships between pixels in an image. The image can be made larger or smaller. It can be rotated, shifted, or otherwise stretched in a variety of ways. Geometric transformations are used to create thumbnail views, adapt digital video from one playback resolution to another, correct distortions caused by viewing geometry, and align multiple images of the same scene or object.

In this chapter we explore the central concepts behind the geometric transformation of images, including geometric coordinate mappings, image interpolation, and inverse mapping. We show how to apply these techniques using Image Processing Toolbox functions, and we explain underlying toolbox conventions. We conclude the chapter with a discussion of image registration, the process of aligning multiple images of the same scene or object for the purpose of visualization or quantitative comparison.

6.1 Transforming Points

Suppose that (w, z) and (x, y) are two spatial coordinate systems, called the *input space* and *output space*, respectively. A geometric coordinate transformation can be defined that maps input space points to output space points:

$$(x, y) = T\{(w, z)\}$$

where $T\{\cdot\}$ is called a *forward transformation*, or *forward mapping*. If $T\{\cdot\}$ has an inverse, then that inverse maps output space points to input space points:

$$(w, z) = T^{-1}\{(x, y)\}$$

where $T^{-1}\{\cdot\}$ is called the *inverse transformation*, or *inverse mapping*. Figure 6.1 shows the input and output spaces, and it illustrates the forward and inverse transformation for this simple example:

$$(x, y) = T\{(w, z)\} = (w/2, z/2)$$
$$(w, z) = T^{-1}\{(x, y)\} = (2x, 2y)$$

Geometric transformations of images are defined in terms of geometric coordinate transformations. Let $f(w, z)$ denote an image in the input space. We can define a transformed image in the output space, $g(x, y)$, in terms of $f(w, z)$ and $T^{-1}\{\cdot\}$, as follows:

$$g(x, y) = f(T^{-1}\{(x, y)\})$$

Figure 6.2 shows what happens to a simple image when transformed using $(x, y) = T\{(w, z)\} = (w/2, z/2)$. This transformation shrinks the image to half its original size.

The Image Processing Toolbox represents geometric coordinate transformations using a so-called *tform structure*, which is created using function `maketform`. The calling syntax for `maketform` is




```
tform = maketform(transform_type, params, ...)
```

The first argument, `transform_type`, is one of the following strings: `'affine'`, `'projective'`, `'custom'`, `'box'`, or `'composite'`. Additional arguments depend on the transform type and are described in detail in the `maketform` documentation.

In this section our interest is in the `'custom'` transform type, which can be used to create a tform structure based on a user-defined geometric coordinate transformation. (Some of the other transformations are discussed later in this chapter.) The full syntax for the `'custom'` type is

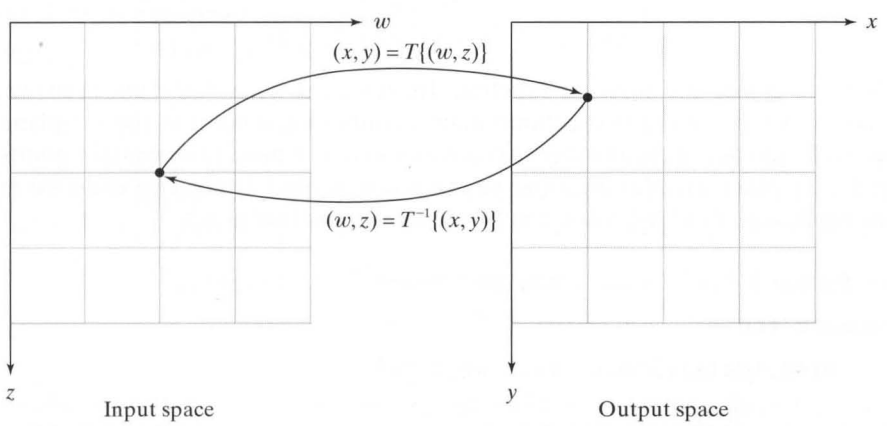


FIGURE 6.1 Forward and inverse transformation of a point for $T\{(w, z)\} = (w/2, z/2)$.

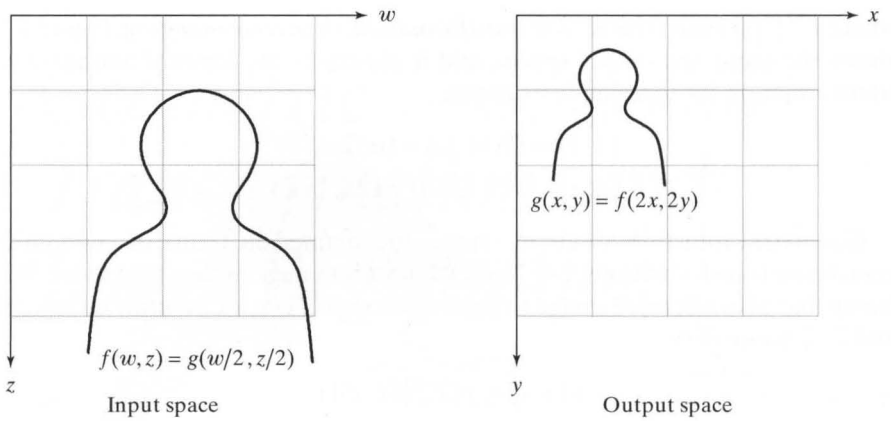


FIGURE 6.2 Forward and inverse transformation of a simple image for the transformation $T\{(w,z)\} = (w/2, z/2)$.

```
tform = maketform('custom', ndims_in, ndims_out, ...
                  forward_fcn, inv_function, tdata)
```

For two-dimensional geometric transformations, `ndims_in` and `ndims_out` are both 2. Parameters `forward_fcn` and `inv_fcn` are function handles for the forward and inverse spatial coordinate transformations. Parameter `tdata` contains any extra information needed by `forward_fcn` and `inverse_fcn`.

EXAMPLE 6.1: Creating a custom tform structure and using it to transform points.

■ In this example we create two tform structures representing different spatial coordinate transformations. The first transformation scales the input horizontally by a factor of 3 and vertically by a factor of 2:

$$(x,y) = T\{(w,z)\} = (3w, 2z)$$
$$(w,z) = T^{-1}\{(x,y)\} = (x/3, y/2)$$

First, we create the forward function. Its syntax is `xy = fwd_function(wz, tdata)`, where `wz` is a two-column matrix containing a point in the wz-plane on each row, and `xy` is another two-column matrix whose rows contain points in the xy-plane. (In this example `tdata` is not needed. It must be included in the input argument list, but it can be ignored in the function.)

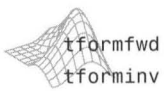
```
>> forward_fcn = @(wz, tdata) [3*wz(:,1), 2*wz(:,2)]
forward_fcn =
    @(wz,tdata)[3*wz(:,1),2*wz(:,2)]
```

Next we create the inverse function having the syntax `wz = inverse_fcn(xy, tdata)`:

```
>> inverse_fcn = @(xy, tdata) [xy(:,1)/3, xy(:,2)/2]
inverse_fcn =
    @(xy,tdata)[xy(:,1)/3,xy(:,2)/2]
```

Now we can make our first tform structure:

```
>> tform1 = maketform('custom', 2, 2, forward_fcn, ...
                      inverse_fcn, [])
tform1 =
       ndims_in: 2
      ndims_out: 2
    forward_fcn: @(wz,tdata)[3*wz(:,1),2*wz(:,2)]
    inverse_fcn: @(xy,tdata)[xy(:,1)/3,xy(:,2)/2]
          tdata: []
```



The toolbox provides two functions for transforming points: `tformfwd` computes the forward transformation, $T\{(w,z)\}$, and `tforminv` computes the inverse transformation, $T^{-1}\{(x,y)\}$. The calling syntaxes are `XY = tformfwd(WZ, tform)` and `WZ = tforminv(XY, tform)`. Here, `WZ` is a $P \times 2$ matrix of points; each row of `WZ` contains the w and z coordinates of one point. Similarly, `XY` is a $P \times 2$ matrix of points containing a pair of x and y coordinates on each row.

For example, the following commands compute the forward transformation of a pair of points, followed by the inverse transformation to verify that we get back the original data:

```
>> WZ = [1 1; 3 2];
>> XY = tformfwd(WZ, tform1)
XY =
    3    2
    9    4
>> WZ2 = tforminv(XY, tform1)
WZ2 =
    1    1
    3    2
```

Our second transformation example shifts the horizontal coordinates as a factor of the vertical coordinates, and leaves the vertical coordinates unchanged.

$$(x,y) = T\{(w,z)\} = (w + 0.4z, z)$$
$$(w,z) = T^{-1}\{(x,y)\} = (x - 0.4y, y)$$

```
>> forward_fcn = @(wz, tdata) [wz(:,1) + 0.4*wz(:,2), ...
                               wz(:, 2)];
>> inverse_fcn = @(xy, tdata) [xy(:,1) - 0.4*xy(:,2), ...
```

```
                                    xy(:,2)];
>> tform2 = maketform('custom', 2, 2, forward_fcn, ...
                      inverse_fcn, []);
>> XY = tformfwd(WZ, tform2)
XY =
    1.4000    1.0000
    3.8000    2.0000
>> WZ2 = tforminv(XY, tform2)
WZ2 =
    1.0000    1.0000
    3.0000    2.0000
```

As you can see, the second column of XY, which corresponds to vertical coordinates, was unchanged by the transformation. ■

To get a better feel for the effects of a particular spatial transformation, it helps to visualize the transformation effect on a set of points arranged on a grid. The following two custom M-functions, `pointgrid` and `vistform`, help visualize a given transformation. Function `pointgrid` constructs a set of grid points to use for the visualization. Note the combined use of functions `meshgrid` (see Section 2.10.5) and `linspace` (see Section 2.8.1) for creating the grid.

pointgrid

```
function wz = pointgrid(corners)
%POINTGRID Points arranged on a grid.
%   WZ = POINTGRID(CORNERS) computes a set point of points on a
%   grid containing 10 horizontal and vertical lines.  Each line
%   contains 50 points.  CORNERS is a 2-by-2 matrix.  The first
%   row contains the horizontal and vertical coordinates of one
%   corner of the grid. The second row contains the coordinates
%   of the opposite corner.  Each row of the P-by-2 output
%   matrix, WZ, contains the coordinates of a point on the output
%   grid.

% Create 10 horizontal lines containing 50 points each.
[w1, z1] = meshgrid(linspace(corners(1,1), corners(2,1), 46), ...
    linspace(corners(1), corners(2), 10));

% Create 10 vertical lines containing 50 points each.
[w2, z2] = meshgrid(linspace(corners(1), corners(2), 10), ...
    linspace(corners(1), corners(2), 46));

% Create a P-by-2 matrix containing all the input-space points.
wz = [w1(:) z1(:); w2(:) z2(:)];
```

The next M-function, `vistform`, transforms a set of input points, and then plots the input points in input space, as well as the corresponding transformed

points in output space. It adjusts the axes limits on both plots to make it easy to compare them.

vistform

```
function vistform(tform, wz)
%VISTFORM Visualization transformation effect on set of points.
%   VISTFORM(TFORM, WZ) shows two plots.  On the left are the
%   points in each row of the P-by-2 matrix WZ.  On the right are
%   the spatially transformed points using TFORM.

% Transform the points to output space.
xy = tformfwd(tform, wz);

% Compute axes limits for both plots.  Bump the limits outward
% slightly.
minlim = min([wz; xy], [], 1);
maxlim = max([wz; xy], [], 1);
bump = max((maxlim - minlim) * 0.05, 0.1);
limits = [minlim(1)-bump(1), maxlim(1)+bump(1), ...
    minlim(2)-bump(2), maxlim(2)+bump(2)];

subplot(1,2,1)
grid_plot(wz, limits, 'w', 'z')

subplot(1,2,2)
grid_plot(xy, limits, 'x', 'y')

%-----------------------------------------------------------------%
function grid_plot(ab, limits, a_label, b_label)
plot(ab(:,1), ab(:,2), '.', 'MarkerSize', 2)
axis equal, axis ij, axis(limits);
set(gca, 'XAxisLocation', 'top')
xlabel(a_label), ylabel(b_label)
```

These functions can be used to visualize the effects of the two spatial transformations we defined in Example 6.1.

```
>> vistform(tform1, pointgrid([0 0;100 100]))
>> figure, vistform(tform2, pointgrid([0 0;100 100]))
```

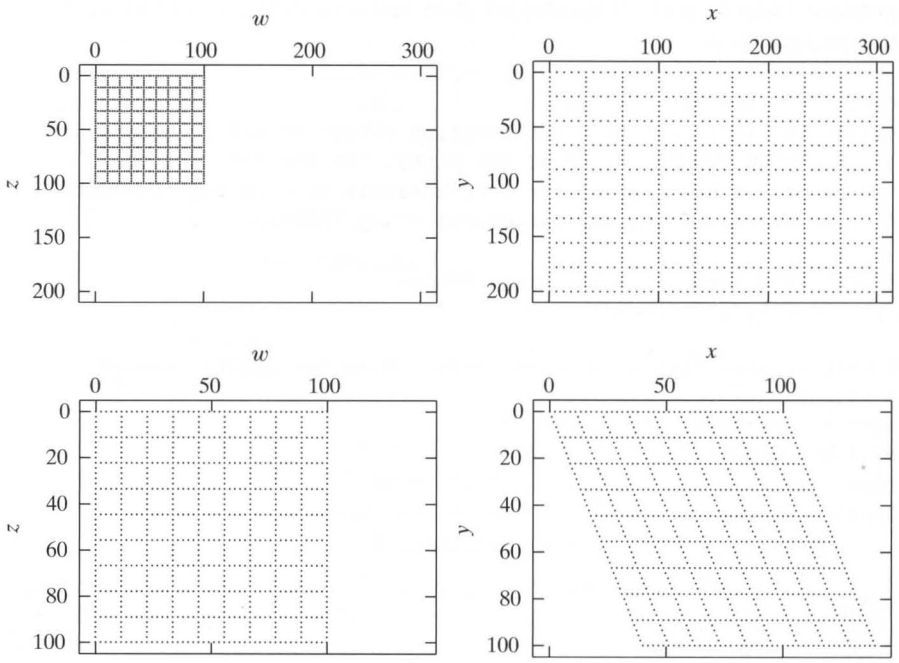
Figure 6.3 shows the results. The first transformation, shown in Fig. 6.3(a) and (b), stretches horizontally and vertically by different scale factors. The second transformation, shown in Fig. 6.3(c) and (d), shifts points horizontally by an amount that varies with the vertical coordinate. This effect is called *shearing*.

6.2 Affine Transformations

Example 6.1 in the previous section shows two *affine transformations*. An affine transformation is a mapping from one vector space to another, consisting of a linear part, expressed as a matrix multiplication, and an additive part, an

a b
c d

FIGURE 6.3
Visualizing the effect of spatial transformations on a grid of points. (a) Grid 1. (b) Grid 1 transformed using `tform1`. (c) Grid 2. (d) Grid 2 transformed using `tform2`.



offset or translation. For two-dimensional spaces, an affine transformation can be written as

$$[x\ y] = [w\ z]\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + [b_1\ b_2]$$

As a mathematical and computational convenience, the affine transformation can be written as a single matrix multiplication by adding a third coordinate.

$$[x\ y\ 1] = [w\ z\ 1]\begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ b_1 & b_2 & 1 \end{bmatrix}$$

This equation can be written also as

$$[x\ y\ 1] = [w\ z\ 1]\mathbf{T}$$

where $\mathbf{T}$ is called an *affine matrix*. The notational convention of adding a 1 to the $[x\ y]$ and $[w\ z]$ vectors results in *homogeneous coordinates* (Foley et al. [1995]).

The affine matrix corresponding to `tform1` in Example 6.1 is

$$\mathbf{T} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The affine matrix corresponding to `tform2` is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0.4 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Function `maketform` can create a tform structure directly from an affine matrix using the syntax `tform = maketform('affine', T)`. For example,

```
>> T = [1 0 0; 0.4 1 0; 0 0 1];
>> tform3 = maketform('affine', T);
>> WZ = [1 1; 3 2];
>> XY = tformfwd(WZ, tform3)
XY =
    1.4000    1.0000
    3.8000    2.0000
```

Important affine transformations include scaling, rotation, translation, shearing, and reflection. Table 6.1 shows how to choose values for the affine matrix, **T**, to achieve these different kinds of transformations.

Several of these types, including rotation, translation, and reflection, belong to an important subset of affine transformations called *similarity transformations*. A similarity transformation preserves angles between lines and changes all distances in the same ratio. Roughly speaking, a similarity transformation *preserves shape*.

An affine transformation is a similarity transformation if the affine matrix has one of the following forms:

$$\mathbf{T} = \begin{bmatrix} s\cos\theta & s\sin\theta & 0 \\ -s\sin\theta & s\cos\theta & 0 \\ b_1 & b_2 & 1 \end{bmatrix}$$

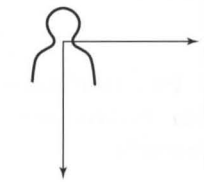
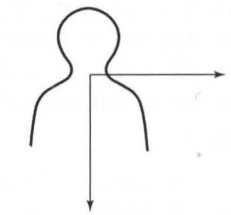
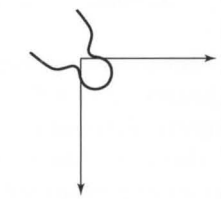
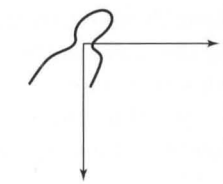
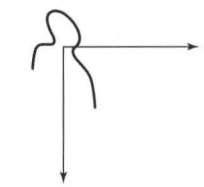
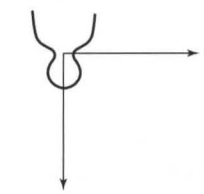
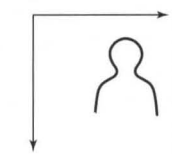
or

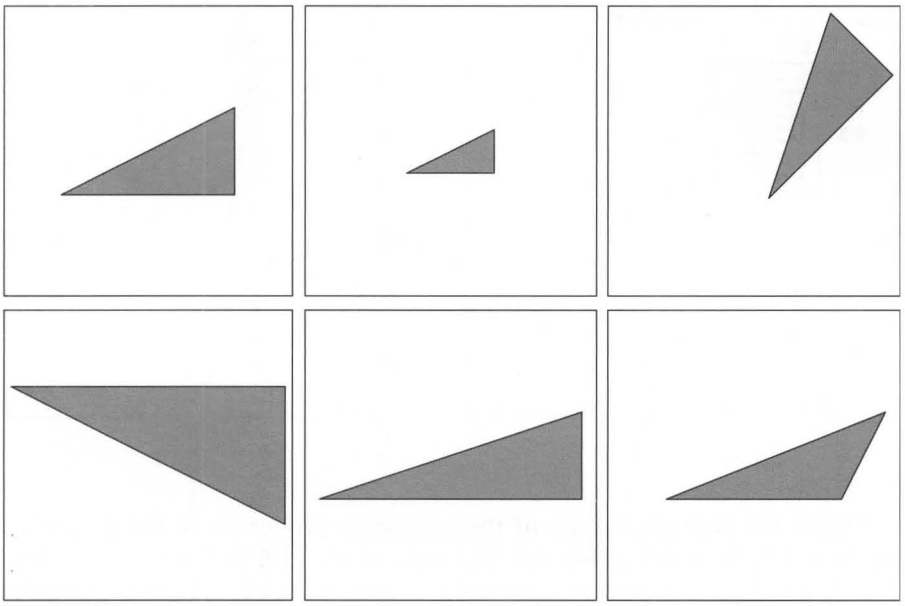
$$\mathbf{T} = \begin{bmatrix} s\cos\theta & s\sin\theta & 0 \\ s\sin\theta & -s\cos\theta & 0 \\ b_1 & b_2 & 1 \end{bmatrix}$$

Note that scaling is a similarity transformation when the horizontal and vertical scale factors are the same.

Similarity transformations can be useful in image processing applications involving solid, relatively flat objects. Images of such objects as they move, or rotate, or as the camera moves closer or further away, are related by similarity transformations. Figure 6.4 shows several similarity transformations applied to a triangular object.

TABLE 6.1 Types of affine transformations.

Type	Affine Matrix, T	Coordinate Equations	Diagram
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = z$	
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = s_x w$ $y = s_y z$	
Rotation	$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w\cos\theta - z\sin\theta$ $y = w\sin\theta + z\cos\theta$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w + \alpha z$ $y = z$	
Shear (vertical)	$\begin{bmatrix} 1 & \beta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = \beta w + z$	
Vertical reflection	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = -z$	
Translation	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{bmatrix}$	$x = w + \delta_x$ $y = z + \delta_y$	



a b c
d e f

FIGURE 6.4 Examples of similarity transformations. (a) Original object. (b) Scaled. (c) Rotated and translated. (d) Reflected and scaled. (e) Scaled horizontally but not vertically—not a similarity. (f) Horizontal shearing—not a similarity.

6.3 Projective Transformations

Another useful geometric transformation type is the *projective transformation*. Projective transformations, which include affine transformations as a special case, are useful for reversing perspective distortion in an image. As with affine transformations, it is useful to define two-dimensional projective transformations using an auxiliary third dimension. Unlike for affine transformations, however, the auxiliary coordinate (denoted by h in the following equation) is not a constant:

$$[x'\ y'\ h] = [w\ z\ 1]\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ b_1 & b_2 & 1 \end{bmatrix}$$

where a_{13} and a_{23} are nonzero, and where $x = x'/h$ and $y = y'/h$. In a projective transformation, lines map to lines but most parallel lines do not stay parallel.

To create a projective tform structure, use the `'projective'` transform type with the `maketform` function. For example,

```
>> T = [-2.7390  0.2929 -0.6373
        0.7426 -0.7500  0.8088
        2.8750  0.7500  1.0000];
>> tform = maketform('projective', T);
>> vistform(tform, pointgrid([0 0; 1 1]));
```

Figure 6.5 illustrates the effect of this projective transformation.

a b

FIGURE 6.5 Example of a projective transformation. (a) Point grid in input space. (b) Transformed point grid in output space.

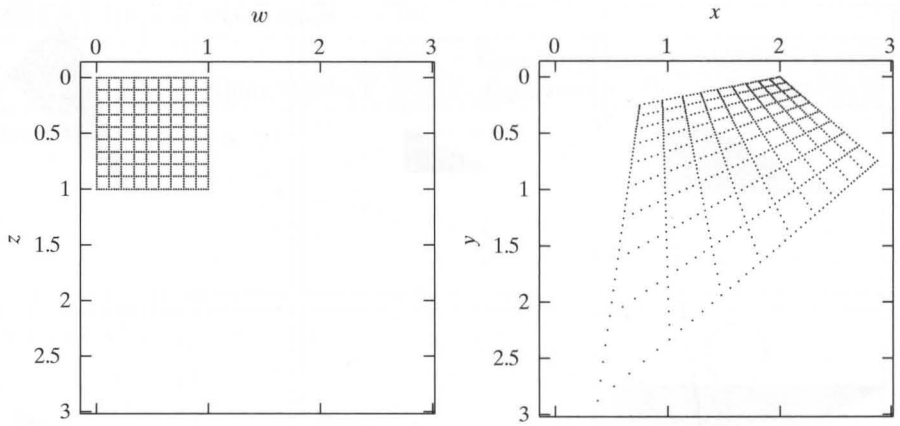


Figure 6.6 illustrates some of the geometric properties of the projective transformation shown in Fig. 6.5. The input-space grid in Fig. 6.5(a) has two sets of parallel lines, one vertical and the other horizontal. Figure 6.6 shows that these sets of parallel lines transform to output-space lines that intersect at locations called *vanishing points*. Vanishing points lie on the *horizon line*. Only input-space lines parallel to the horizon line remain parallel when transformed. All other sets of parallel lines transform to lines that intersect at a vanishing point on the horizon line.

6.4 Applying Geometric Transformations to Images

Now that we have seen how to apply geometric transformations to points, we can proceed to consider how to apply them to images. The following equation from Section 6.1 suggests a procedure:

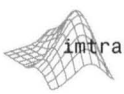
$$g(x, y) = f(T^{-1}\{(x, y)\})$$

The procedure for computing the output pixel at location (x_k, y_k) is:

1. Evaluate $(w_k, z_k) = T^{-1}\{(x_k, y_k)\}$.
2. Evaluate $f(w_k, z_k)$.
3. $g(x_k, y_k) = f(w_k, z_k)$.

We will have more to say about step 2 in Section 6.6, when we discuss image interpolation. Note how this procedure uses only the inverse spatial transformation, $T^{-1}\{\cdot\}$, and not the forward transformation. For this reason, the procedure is often called *inverse mapping*.

The Image Processing Toolbox function `imtransform` uses the inverse mapping procedure to apply a geometric transformation to an image. The basic calling syntax for `imtransform` is:




```
g = imtransform(f, tform)
```

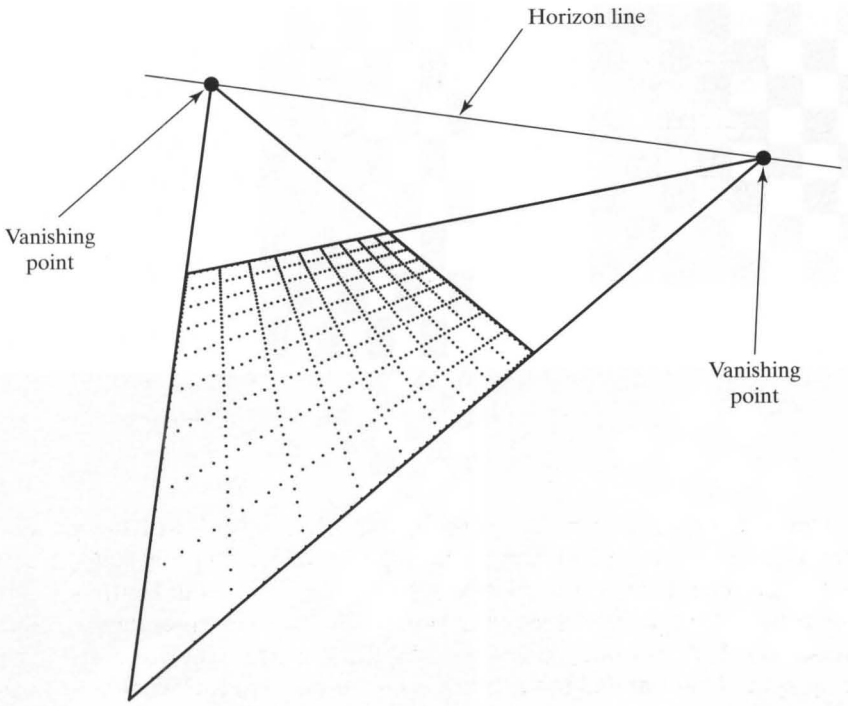


FIGURE 6.6 Vanishing points and the horizon line for a projective transformation.

EXAMPLE 6.2: Geometric transformations of images.

■ In this example we use functions `checkerboard` and `imtransform` to explore different spatial transformations on images. As shown in Table 6.1, an affine transformation for scaling an image has the form

$$\mathbf{T} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

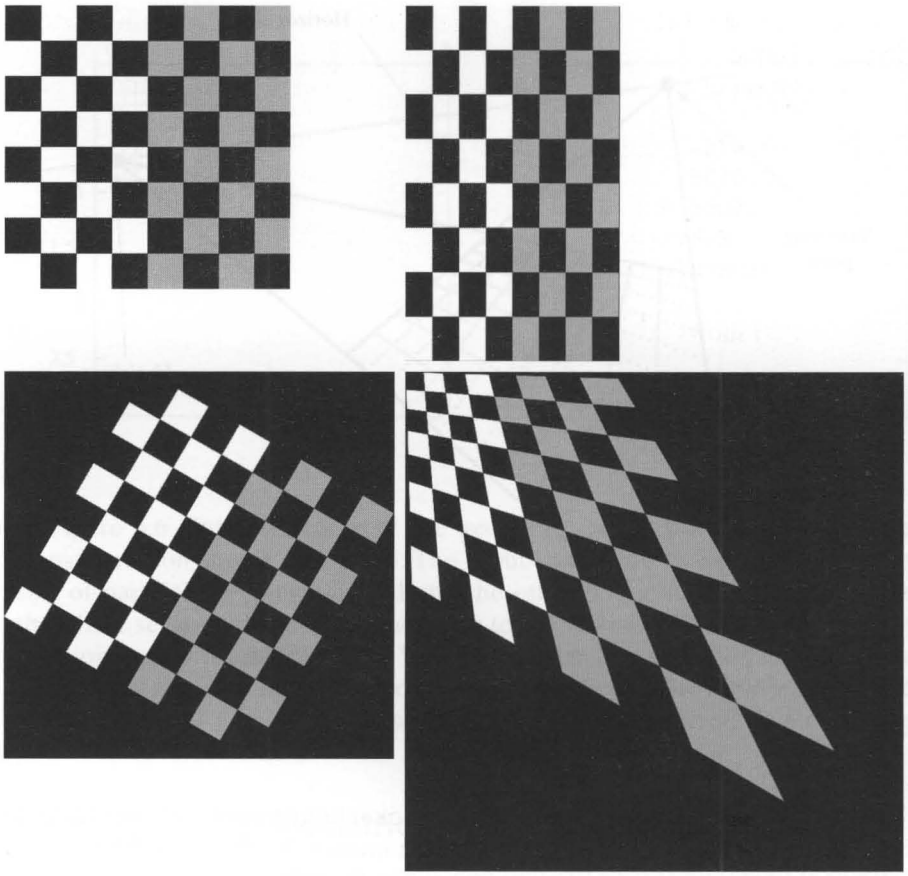
The following commands generate a scaling tform structure and apply it to a checkerboard test image.

See Section 5.4 regarding function `checkerboard`.

```
>> f = checkerboard(50);
>> sx = 0.75;
>> sy = 1.25;
>> T = [sx 0  0
        0  sy 0
        0  0  1];
>> t1 = maketform('affine', T);
>> g1 = imtransform(f, t1);
```

Figures 6.7(a) and (b) show the original and scaled checkerboard images.

An affine matrix for rotation has the form



a b
c d

FIGURE 6.7
Geometric transformations of the checkerboard image. (a) Original image. (b) Affine scaling transformation. (c) Affine rotation transformation. (d) Projective transformation.

$$T = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The following commands rotate the test image using an affine transformation:

```
>> theta = pi/6;
>> T2 = [ cos(theta) sin(theta) 0
         -sin(theta) cos(theta) 0
          0          0          1];
>> t2 = maketform('affine', T2);
>> g2 = imtransform(f, t2);
```

Figure 6.7(c) shows the rotated image. The black regions of the output image correspond to locations outside the bounds of the input image; `imtransform` sets these pixels to 0 (black) by default. See Examples 6.3 and 6.4 for a method to use a color other than black. It it worth noting that Image Processing Tool-

box function `imrotate` (see Section 12.4.3) is based on the procedure outlined in this example.

The next set of commands demonstrate a projective transformation.

```
>> T3 = [0.4788 0.0135 -0.0009
        0.0135 0.4788 -0.0009
        0.5059 0.5059  1.0000];
>> tform3 = maketform('projective', T3);
>> g3 = imtransform(f, tform3);
```

Figure 6.7(d) shows the result. ■

6.5 Image Coordinate Systems in MATLAB

Before considering other aspects of geometric transformations in MATLAB, we pause to revisit the issue of how MATLAB displays image coordinates.

You should review Section 2.1.1, in which we discuss the axis convention we use in the book, and compare that convention to the convention used by the toolbox, and by MATLAB.

Figure 6.7, like many other figures in this book, shows images without axes ticks and labels. That is the default behavior of function `imshow`. As you will note in the following discussion, however, analysis and interpretation of geometric image transformations are aided significantly by displaying these visual queues.

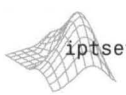
One way to turn on tick labels is to call `axis on` after calling `imshow`. For example,

```
>> f = imread('circuit-board.tif');
>> imshow(f)
>> axis on
>> xlabel x
>> ylabel y
```

Figure 6.8 shows a screen shot of the result. The origin is at the upper left. The x-axis is horizontal and increases to the right. The y-axis is vertical and increases downward. As you will recall, this convention is what we referred to as the image *spatial coordinate system* in Section 2.1.1. The x- and y-axes in this system are the reverse of the book image coordinate system (see Fig. 2.1).

The toolbox function `iptsetpref`, which sets certain user preferences, can be used to make `imshow` display tick labels all the time. To turn on tick-label display, call

```
>> iptsetpref imshowAxesVisible on
```



To make this setting persist from session to session, place the preceding call in your `startup.m` file. (Search for "startup.m" in the MATLAB Help Browser for more details.)

Figure 6.9 examines the image spatial coordinate system more closely for an image with three rows and four columns. The center of the upper-left pixel is located at (1, 1) on the xy-plane. Similarly, the center of the lower-right pixel

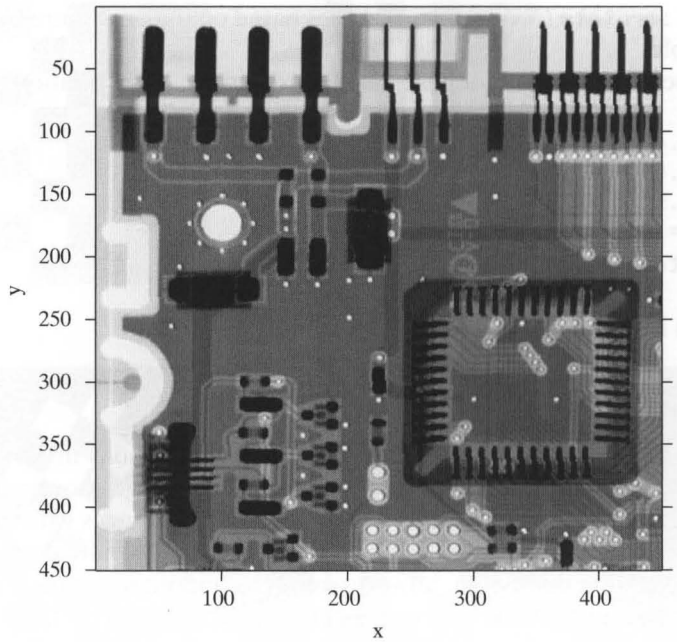


FIGURE 6.8
Image displayed with axes ticks and labels visible using `imshow` and `axis on`. The origin is at the top, left.

is located at (4, 3) on the plane. Each pixel covers a unit area. For example, the upper-left pixel covers the square region from (0.5, 0.5) to (1.5, 1.5).

It is possible to change both the location and the size of image pixels in the xy-plane. This is done by manipulating the `XData` and `YData` properties of the Handle Graphics image object. The `XData` property is a two-element vector in which first element specifies the x-coordinate of the center of the first column of pixels and the second specifies the x-coordinate of the last column of pixels. Similarly, the two elements of the `YData` vector specify the y-coordinates of the centers of the first and last rows.

For an image containing M rows and N columns, the default `XData` vector is `[1 N]` and the default `YData` vector is `[1 M]`. For a 3×4 image, for example, `XData` is `[1 4]` and `YData` is `[1 3]`, which are consistent with the coordinates shown in Figure 6.9.

You can set the `XData` and `YData` properties to other values, which can be very useful when working with geometric transformations. The `imshow` function supports this capability through the use of optional parameter-value pairs. For instance, using the following syntax displays the circuit board image so that the left and right pixels are centered at −20 and 20 on the x-axis, and the top and bottom pixels are centered at −10 and 10 on the y-axis.

```
>> imshow(f, 'XData', [-20 20], 'YData', [-10 10])
>> axis on
>> xlabel x
>> ylabel y
```

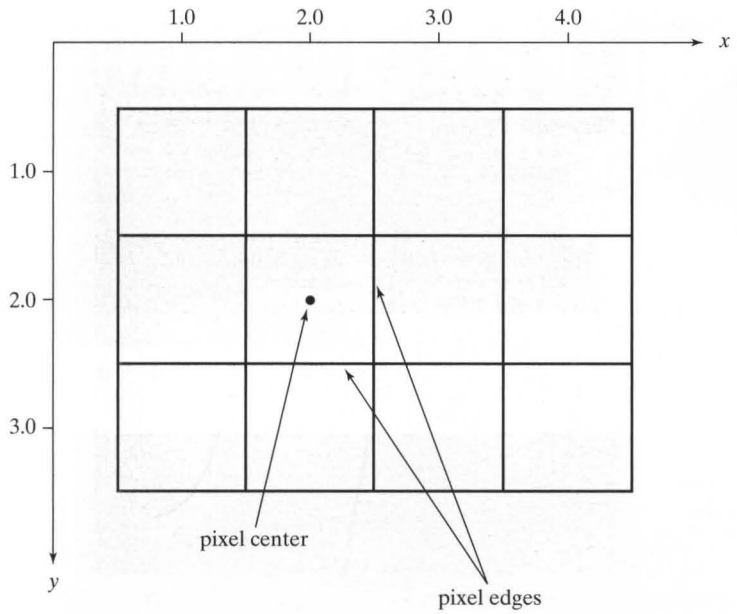


FIGURE 6.9 Spatial coordinate system for image pixels.

Figure 6.10(a) shows the result. Figure 6.10(b) shows the result of zooming in on the upper-left corner of the image using the command

```
>> axis([8 8.5 0.8 1.1])
```

Observe that the pixels in Fig. 6.10(b) are not square.

6.5.1 Output Image Location

Figure 6.7(c), discussed in Example 6.2, shows an image rotated using an affine transformation. Note, however, that the figure does not show the location of the image in output space. Function `imtransform` can provide this information through the use of additional output arguments. The calling syntax is

```
[g, xdata, ydata] = imtransform(f, tform)
```

The second and third output arguments can be used as `XData` and `YData` parameters when displaying the output image using `imshow`. The following example shows how to use these output arguments to display the input and output images together in the same coordinate system.

EXAMPLE 6.3: Displaying input and output images together in a common coordinate system.

■ In this example we use a rotation and a translation to explore how to locate and display the output image in a common coordinate system with the input image. We start by displaying the original image with axes ticks and labels.

```
>> imshow(f)
>> axis on
```

a
b

FIGURE 6.10 (a) Image displayed with nondefault spatial coordinates. (b) Zoomed view of image.

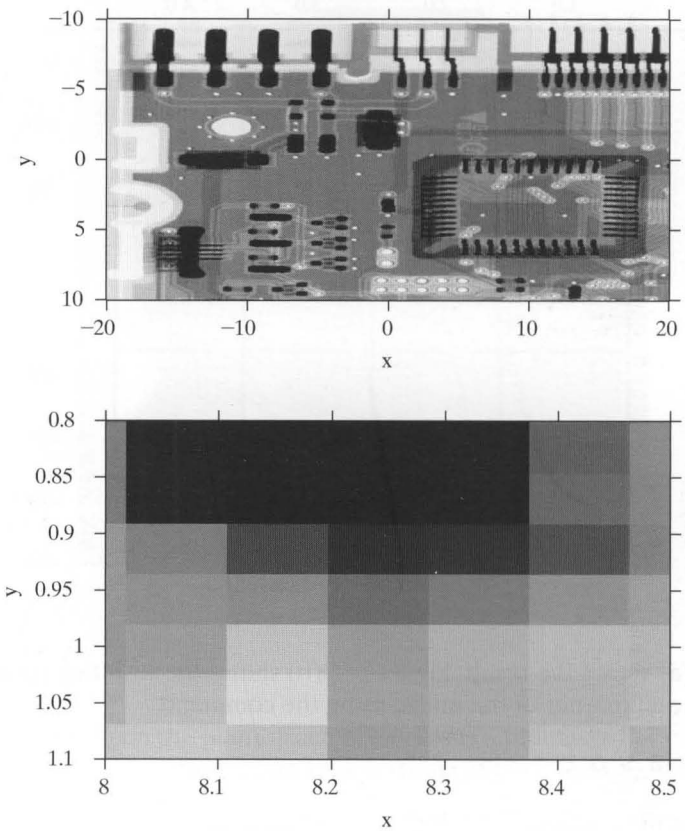
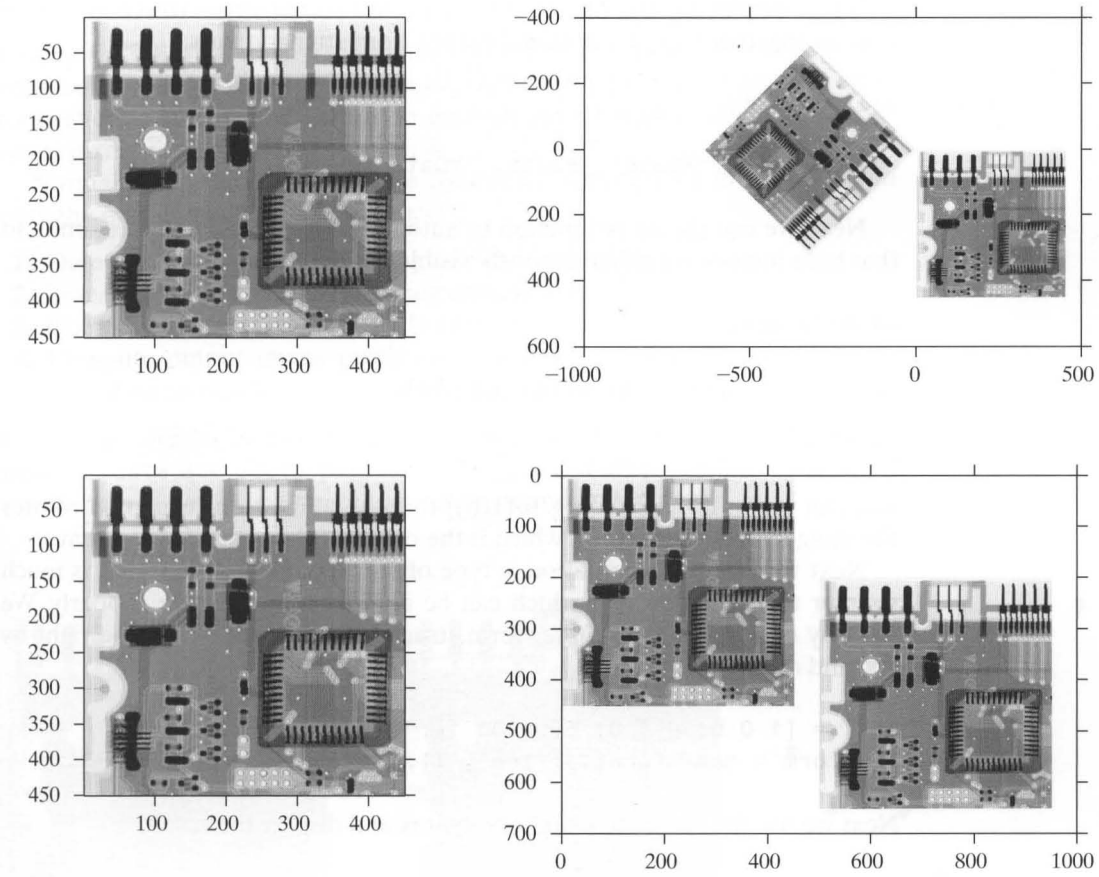


Figure 6.11(a) shows the original image.

Next we use `imtransform` to rotate the image by $3\pi/4$ radians.

```
>> theta = 3*pi/4;
>> T = [ cos(theta) sin(theta) 0
        -sin(theta) cos(theta) 0
         0          0          1];
>> tform = maketform('affine', T);
>> [g, xdata, ydata] = imtransform(f, tform, ...
                                   'FillValue', 255);
```

The call to `imtransform` in the preceding line of code shows two new concepts. The first is the use of the optional output arguments, `xdata` and `ydata`. These serve to locate the output image in the xy-coordinate system. The other concept is the optional input arguments: `'FillValue', 255`. The `FillValue` parameter specifies the value to be used for any output image pixel that corresponds to an input-space location outside the boundaries of the input



a b
c d

FIGURE 6.11 (a) Original image. (b) Original and rotated image displayed using common coordinate system. (c) Translated image as computed using basic `imtransform` syntax. (d) Original and translated image displayed using common coordinate system.

image. By default, this value is 0. That is the reason why the pixels surrounding the rotated checkerboard, in Figure 6.7(c) are black, as mentioned earlier. In this example we want them to be white.

Next we want to display both images at the same time and in a common coordinate system. We follow the usual MATLAB pattern for superimposing two plots or image displays in the same figure:

1. Create the first plot or image display.
2. Call `hold on`, so that subsequent plotting or display commands do not clear the figure.
3. Create the second plot or image display.

When displaying the output image, we use the `XData` / `YData` syntax of `imshow` together with the optional output from `imtransform`:

```
>> imshow(f)
>> hold on
>> imshow(g, 'XData', xdata, 'YData', ydata)
```

Next, we use the `axis` function to automatically expand the axes limits so that both images are simultaneously visible.

```
>> axis auto
```

Finally, we turn on the axes ticks and labels.

```
>> axis on
```

You can see in the result [Fig. 6.11(b)] that the affine transformation rotates the image about point (0, 0), which is the origin of the coordinate system.

Next we examine translation, a type of affine transformation that is much simpler than rotation, but which can be confusing to visualize properly. We start by constructing an affine tform structure that translates to the right by 500 and down by 200.

```
>> T = [1 0 0; 0 1 0; 500 200 1];
>> tform = maketform('affine', T);
```

Next we use the basic `imtransform` syntax and display the result.

```
>> g = imtransform(f, tform);
>> imshow(g)
>> axis on
```

Figure 6.11(c) shows the result, which puzzlingly looks exactly like the original image in Fig. 6.11(a). The explanation for this mystery is that `imtransform` automatically captures just enough pixels in output space to show only the transformed image. This automatic behavior effectively eliminates the translation.

To see clearly the translation effect, we use the same technique that we used above for rotation:

```
>> [g, xdata, ydata] = imtransform(f, tform, ...
                                   'FillValue', 255);
>> imshow(f)
>> hold on
>> imshow(g, 'XData', xdata, 'YData', ydata)
>> axis on
>> axis auto
```

Figure 6.11(d) shows the result. ■

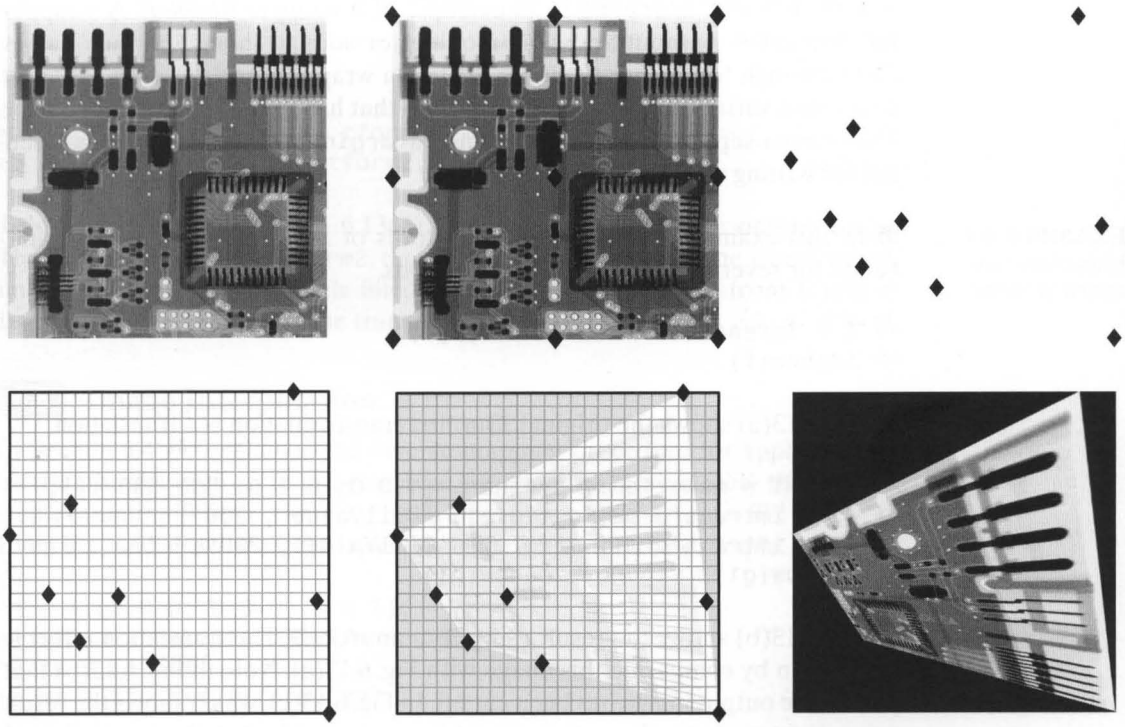
6.5.2 Controlling the Output Grid

Example 6.3 illustrated how to visualize the effect of a translation by using the `xdata` and `ydata` parameters, which are output from `imtransform` and input to `imshow`. Another approach is to exercise direct control over the output-space pixel grid used by `imtransform`.

Normally, `imtransform` uses the following procedure to locate and compute the output image in output space:

1. Determine the bounding rectangle of the input image.
2. Transform points on the bounding rectangle into output space.
3. Compute the bounding rectangle of the transformed output-space points.
4. Compute output image pixels on a grid lying within the output-space bounding rectangle.

Figure 6.12 illustrates this procedure. The procedure can be customized by passing `xdata` and `ydata` parameters into `imtransform`, which uses these parameters to determine the output-space bounding rectangle.



a b c
d e f

FIGURE 6.12 (a) Original image. (b) Point grid along edges and in center of image. (c) Transformed point grid. (d) Bounding box of transformed point grid, with output pixel grid. (e) Output image pixels computed inside automatically-determined output pixel grid. (f) Final result.

The custom function listed below illustrates this use of the `xdata` and `ydata` parameters. It is a variation of `imtransform` that always uses the input-space rectangle as the output-space rectangle. That way, the positions of the input and output images can be compared more directly.

`imtransform2`

```
function g = imtransform2(f, varargin)
%IMTRANSFORM2 2-D image transformation with fixed output location
%   G = IMTRANSFORM2(F, TFORM, ...) applies a 2-D geometric
%   transformation to an image.  IMTRANSFORM2 fixes the output image
%   location to cover the same region as the input image.
%   IMTRANSFORM2 takes the same set of optional parameter/value
%   pairs as IMTRANSFORM.

[M, N] = size(f);
xdata = [1 N];
ydata = [1 M];
g = imtransform(f, varargin{:}, 'XData', xdata, ...
                'YData', ydata);
```

Function `imtransform2` is an example of a *wrapper function*. A wrapper function takes its inputs, possibly modifies or adds to them, and then passes them through to another function. Writing a wrapper function is an easy way to create a variation of an existing function that has different default behavior. The comma-separated list syntax using `varargin` (see Section 3.2.4) is essential for writing wrapper functions easily.

EXAMPLE 6.4: Using function `imtransform2`.

■ In this example we compare the outputs of `imtransform` and `imtransform2` for several geometric transformations.

```
>> f = imread('lunar-shadows.jpg');
>> imshow(f)
```

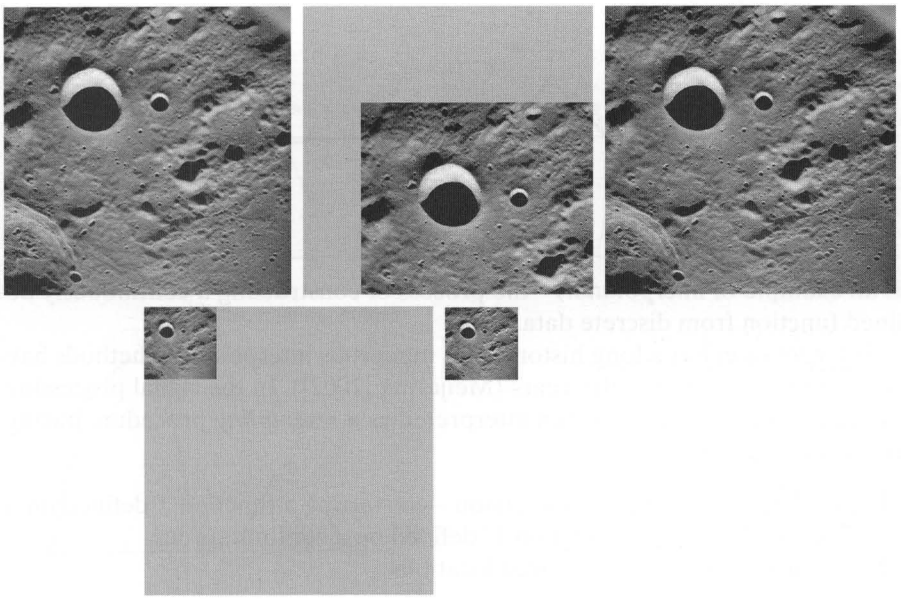
Figure 6.13(a) shows the original. Our first transformation is a translation.

```
>> tform1 = maketform('affine', [1 0 0; 0 1 0; 300 500 1]);
>> g1 = imtransform2(f, tform1, 'FillValue', 200);
>> h1 = imtransform(f, tform1, 'FillValue', 200);
>> imshow(g1), figure, imshow(h1)
```

Figure 6.13(b) shows the result using `imtransform2`. The translation effect is easily seen by comparing this image with Fig. 6.13(a). Note in Fig. 6.13(b) that part of the output image has been cut off. In Fig. 6.13(c), which shows the result using `imtransform`, the entire output image is visible, but the translation effect has been lost.

Our second transformation shrinks the input by a factor of 4 in both directions.

```
>> tform2 = maketform('affine', [0.25 0 0; 0 0.25 0; 0 0 1]);
```



a b c
d e

FIGURE 6.13
(a) Original image. (b) Translation using function `imtransform2`. (c) Translation using `imtransform` and default parameters. (d) Scaling using `imtransform2`. (e) Scaling using `imtransform` and default parameters. (Original image courtesy of NASA.)

```
>> g2 = imtransform2(f, tform2, 'FillValues', 200);
>> h2 = imtransform(f, tform2, 'FillValues', 200);
```

This time, both outputs [Fig. 6.13(d) and (e)] show the entire output image. The output from `imtransform2`, though is much bigger than the transformed image, with the "extra" pixels filled in with gray. The output from function `imtransform` contains just the transformed image. ■

6.6 Image Interpolation

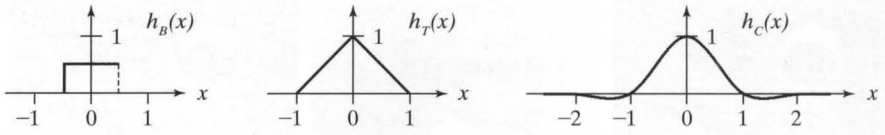
In Section 6.4 we explained the inverse mapping procedure for applying geometric transformations to images. Here, we examine more closely the second step, evaluating $f(w_k, z_k)$, where f is the input image and $(w_k, z_k) = T^{-1}\{(x_k, y_k)\}$. Even if x_k and y_k are integers, w_k and z_k usually are not. For example:

```
>> T = [2 0 0; 0 3 0; 0 0 1];
>> tform = maketform('affine', T);
>> xy = [5 10];
>> wz = tforminv(tform, xy)
wz =
    2.5000    3.3333
```

For digital images, the values of f are known only at integer-valued locations. Using these known values to evaluate f at non-integer-valued locations

a b c

FIGURE 6.14
(a) Box, (b) triangle, and (c) cubic interpolation kernels.



is an example of *interpolation*—the process of constructing a continuously defined function from discrete data.

Interpolation has a long history, with numerous interpolation methods having been proposed over the years (Meijering [2002]). In the signal processing literature, interpolation is often interpreted as a *resampling* procedure having two conceptual steps:

1. Discrete to continuous conversion—converting a function f defined on a discrete domain to a function f' defined on a continuous one.
2. Evaluation of f' at the desired locations.

This interpretation is most useful when the known samples of f are spaced regularly. The discrete-to-continuous conversion step can be formulated as a sum of scaled and shifted functions called *interpolation kernels*. Figure 6.14 shows several commonly-used interpolation kernels: the box kernel, $h_B(x)$, the triangle kernel, $h_T(x)$, and the cubic kernel, $h_C(x)$. The box kernel is defined by the equation:

$$h_B(x) = \begin{cases} 1 & -0.5 \le x < 0.5 \\ 0 & \text{otherwise} \end{cases}$$

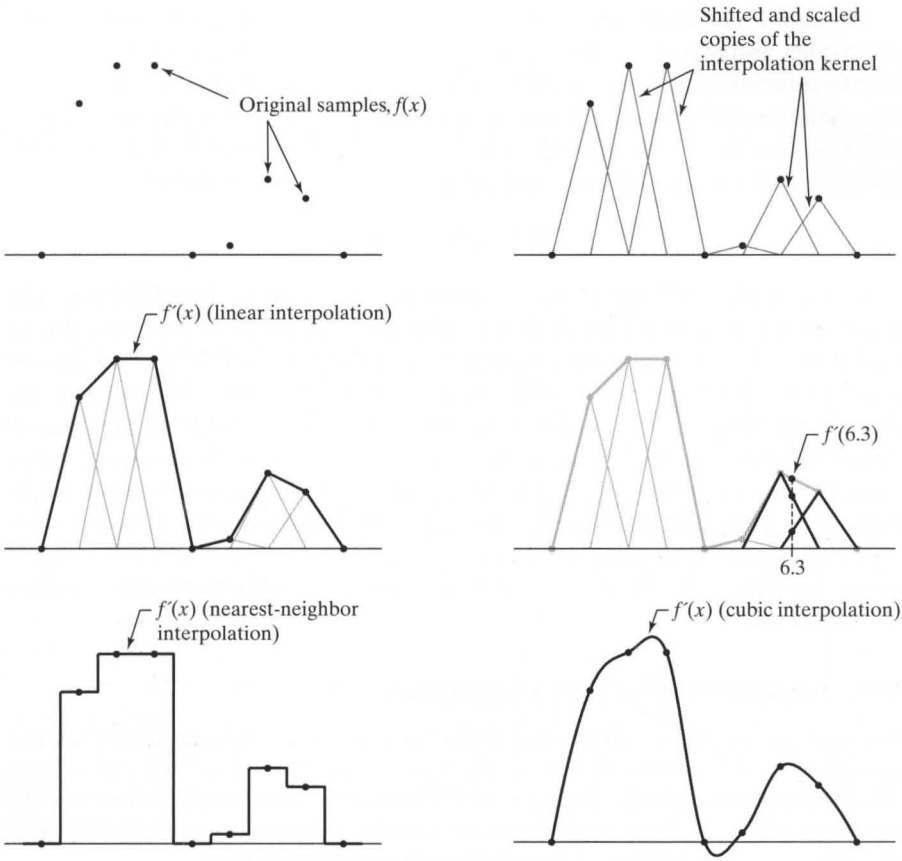
The triangle kernel is defined by the equation:

$$h_T(x) = \begin{cases} 1 - |x| & \text{for } x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

And the cubic kernel is defined by the equation:

$$h_C(x) = \begin{cases} 1.5|x|^3 - 2.5|x|^2 + 1 & |x| \le 1 \\ -0.5|x|^3 + 2.5|x|^2 - 4|x| + 2 & 1 < |x| \le 2 \\ 0 & \text{otherwise} \end{cases}$$

There are other cubic kernels with different coefficients, but the preceding form is the one used most commonly in image processing (Keys [1983]).



a b
c d
e f

FIGURE 6.15 (a) Original samples, $f(x)$. (b) Copies of triangular interpolation kernel, $h_T(x)$, shifted and then scaled by the values of $f(x)$. (c) Sum of the shifted and scaled interpolation kernels, resulting in $f'(x)$ as computed using linear interpolation. (d) Computing $f'(6.3)$ as $f(6)\,h_T(0.3) + f(7)\,h_T(0.7)$. (e) $f'(x)$ computed using nearest-neighbor interpolation. (f) $f'(x)$ computed using cubic interpolation.

Figure 6.15 illustrates how one-dimensional interpolation works. Figure 6.15(a) shows a one-dimensional discrete signal $f(x)$, and Fig. 6.15(b) shows the interpolation kernel $h_T(x)$. In Figure 6.15(c), copies of the kernel are scaled by the values of $f(x)$ and shifted to the corresponding locations. Figure 6.15(d) shows the continuous-domain function, $f'(x)$, which is formed by adding all the scaled and shifted copies of the kernel. Interpolation using triangular kernels, is a form of *linear interpolation* (Gonzalez and Woods [2008]).

As a computational procedure to be implemented in software, the conceptual two-step procedure mentioned earlier is not useful. First, there is no practical way to represent in memory all the values of a continuous-domain function. Second, because only some of the values of $f'(x)$ are actually needed, it would be wasteful to compute them all, even if that were possible. Consequently, in software implementations of interpolation, the entire signal $f'(x)$ is never formed explicitly. Instead, individual values of $f'(x)$ are computed as needed. Figure 6.15(d) shows the method for computing $f'(3.4)$ using the triangular kernel. Only two of the shifted kernels are nonzero at $x = 3.4$, so $f'(3.4)$ is computed as the sum of only two terms: $f(3)h_T(0.4) + f(4)h_T(-0.6)$.

Figure 6.15(e) shows $f'(x)$ computed using the box kernels. It can be shown (Gonzalez and Woods [2008]) that interpolation using box kernels is equivalent to a technique called *nearest-neighbor interpolation*. In nearest neighbor interpolation, the value of $f'(x)$ is computed as the value of $f(y)$ at the location y closest to x. If $f(y)$ is defined for integer values of y, then nearest-neighbor interpolation can be implemented using a simple round operation:

$$f'(x) = f(\text{round}(x))$$

Figure 6.15(e) shows $f'(x)$ as computed using cubic interpolation. The graph shows an important difference in behavior between linear and cubic interpolation. Cubic interpolation exhibits *overshoot* at locations with large differences between adjacent samples of $f(x)$. Because of this phenomenon, the interpolated curve $f'(x)$ can take on values outside the range of the original samples. Linear interpolation, on the other hand, never produces out-of-range values. In image processing applications, overshoot is sometimes beneficial, in that it can have a visual "sharpening" effect that improves the appearance of images. On the other hand, it can be a disadvantage sometimes, for example when it produces negative values in situations where only nonnegative values are expected.

6.6.1 Interpolation in Two Dimensions

The most common two-dimensional interpolation approach used in image processing is to decompose the problem into a sequence of several one-dimensional interpolation tasks. Figure 6.16 illustrates the process with a few specific values, in which $f'(2.6, 1.4)$ is obtained from the surrounding samples of $f(x, y)$ by using a sequence of one-dimensional linear interpolations:

1. Determine $f'(2.6, 1.0)$ by linearly interpolating between $f(2, 1)$ and $f(3, 1)$.
2. Determine $f'(2.6, 2.0)$ by linearly interpolating between $f(2, 2)$ and $f(3, 2)$.
3. Determine $f'(2.6, 1.4)$ by linearly interpolating between $f'(2.6, 1.0)$ and $f'(2.6, 2.0)$.

The process of interpolating in two dimensions using a sequence of one-dimensional linear interpolations is called *bilinear interpolation*. Similarly, *bicubic interpolation* is two-dimensional interpolation performed using a sequence of one-dimensional cubic interpolations.

6.6.2 Comparing Interpolation Methods

Interpolation methods vary in computation speed and in output quality. A classical test used to illustrate the pros and cons of different interpolation methods is repeated rotation. The function listed below uses `imtransform2` to rotate an image 30 degrees about its center point, 12 times in succession. The function forms a geometric transformation that rotates about the center of the image by taking advantage of the composition property of affine transformations. Specifically, if $\mathbf{T}_1$ and $\mathbf{T}_2$ are matrices defining two affine transformations, then the

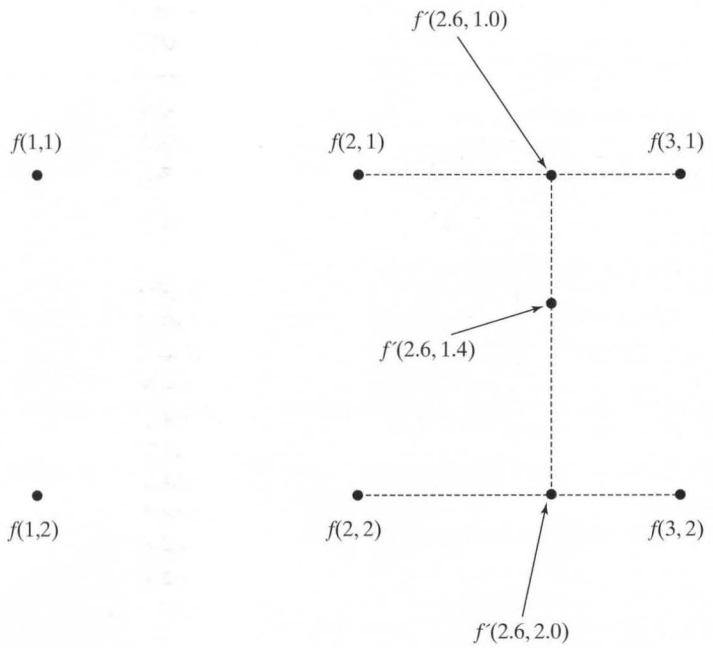


FIGURE 6.16 Computing $f'(2.6, 1.4)$ using bilinear interpolation.

matrix $\mathbf{T} = \mathbf{T}_1\mathbf{T}_2$ defines another affine transformation that is the composition of the first two.

reprotate

```
function g = reprotate(f, interp_method)
%REPROTATE Rotate image repeatedly
%   G = REPROTATE(F, INTERP_METHOD) rotates the input image, F,
%   twelve times in succession as a test of different interpolation
%   methods.  INTERP_METHOD can be one of the strings 'nearest',
%   'bilinear', or 'bicubic'.

% Form a spatial transformation that rotates the image about its
% center point.  The transformation is formed as a composite of
% three affine transformations:
%
% 1. Transform the center of the image to the origin.
center = fliplr(1 + size(f) / 2);
A1 = [1 0 0; 0 1 0; -center, 1];

% 2. Rotate 30 degrees about the origin.
theta = 30*pi/180;
A2 = [cos(theta) -sin(theta) 0; sin(theta) cos(theta) 0; 0 0 1];

% 3. Transform from the origin back to the original center location.
A3 = [1 0 0; 0 1 0; center 1];
```

```
% Compose the three transforms using matrix multiplication.
A = A1 * A2 * A3;
tform = maketform('affine', A);

% Apply the rotation 12 times in sequence. Use imtransform2 so that
% each successive transformation is computed using the same location
% and size as the original image.
g = f;
for k = 1:12
   g = imtransform2(g, tform, interp_method);
end
```

EXAMPLE 6.5: Comparing speed and image quality for several interpolation methods.

■ This example uses `reprotate` to compare computation speed and image quality for nearest neighbor, bilinear, and bicubic interpolation. The function rotates the input 12 times in succession, using the interpolation method specified by the caller.

First, we time each method using `timeit`.

```
>> f = imread('cameraman.tif');
>> timeit(@() reprotate(f, 'nearest'))
ans =
    1.2160
>> timeit(@() reprotate(f, 'bilinear'))
ans =
    1.6083
>> timeit(@() reprotate(f, 'bicubic'))
ans =
    2.3172
```

So nearest-neighbor interpolation is fastest, and bicubic interpolation is slowest, as you would expect.

Next, we evaluate the output image quality.

```
>> imshow(reprotate(f, 'nearest'))
>> imshow(reprotate(f, 'bilinear'))
>> imshow(reprotate(f, 'bicubic'))
```

Figure 6.17 shows the results. The nearest-neighbor result in Fig. 6.17(b) shows significant "jaggy" edge distortion. The bilinear interpolation result in Fig. 6.17(c) has smoother edges but a somewhat blurred appearance overall. The bicubic interpolation result in Fig. 6.17(d) looks best, with smooth edges and much less blurring than the bilinear result. Note that only the central pixels in the image remain in-bounds for all twelve of the repeated rotations. As in Example 6.2, the remaining pixels are black. ■



a b
c d

FIGURE 6.17
Using repeated rotations to compare interpolation methods.
(a) Original image.
(b) Nearest-neighbor interpolation.
(c) Bilinear interpolation.
(d) Bicubic interpolation.
(Original image courtesy of MIT.)

6.7 Image Registration

One of the most important image processing applications of geometric transformations is *image registration*. Image registration methods seek to align two or more images of the same scene. For example, it may be of interest to align images taken at different times. The time difference could be measured in months or years, as with satellite images used to detect environmental changes over long time periods. Or it could be a few weeks, as when using a sequence of medical images to measure tumor growth. The time difference could even be a tiny fraction of a second, as in camera stabilization and target tracking algorithms.

A different scenario arises when multiple images are taken at the same time but with different instruments. For example, two cameras in different positions may acquire simultaneous images of the same scene in order to measure the scene depth.

Sometimes the images come from dissimilar instruments. Two satellite images may differ in both resolution and spectral characteristics. One could be

a high-resolution, visible-light, panchromatic image, and the other could be a low-resolution multispectral image. Or two medical images may be an MRI scan and a PET scan. In these cases the objective is often to *fuse* the disparate images into a single, enhanced visualization of the scene.

In all these cases, combining the images requires compensating for geometric aberrations caused by differences in camera angle, distance, and orientation; sensor resolution; movement of objects in the scene; and other factors.

6.7.1 The Registration Process

Image registration methods generally consist of the following basic steps:

1. Detect features.
2. Match corresponding features.
3. Infer geometric transformation.
4. Use the geometric transformation to align one image with the other.

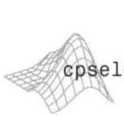
We discuss image features in Chapters 12 and 13.

An image *feature* is any portion of an image that can potentially be identified and located in both images. Features can be points, lines, or corners, for example. Once selected, features have to be matched. That is, for a feature in one image, one must determine the corresponding feature in the other image. Image registration methods can be *manual* or *automatic* depending on whether feature detection and matching is human-assisted or performed using an automatic algorithm.

From the set of matched-feature pairs, a geometric transformation function is inferred that maps features in one image onto the locations of the matching features in the other. Usually a particular parametric transformation model is chosen, based on a particular image capture geometry. For example, assume that two images are taken with the same viewing angle but from a different position, possibly including a rotation about the optical axis. If the scene objects are far enough from the camera to minimize perspective effects, then we can use an affine transformation (Brown [1992]).

An affine transformation is an example of a *global transformation*; that is, the transformation function is the same everywhere in the image. Other global transformation functions commonly used for image registration include projective (see Section 6.3) and polynomial. For many image registration problems, the geometric correspondence between features in the two images is too complex to be characterized by a single transformation function that applies everywhere. For such problems, a transformation functions with locally varying parameters may be used. These functions are called *local transformations*.

6.7.2 Manual Feature Selection and Matching Using `cpselect`



The Image Processing Toolbox uses the term *control points* for image features. The toolbox provides a GUI (graphical user interface) called the Control Point Selection Tool (`cpselect`) for manually selecting and matching corresponding control points in a pair of images to be registered.

The tool is launched by passing the filenames of the images to be aligned as input arguments to `cpselect`. For example,

```
>> cpselect('vector-gis-data.tif', 'aerial-photo-cropped.tif')
```

Alternatively, the images can be read into MATLAB variables first and then passed to `cpselect`:

```
>> f = imread('vector-gis-data.tif');
>> g = imread('aerial-photo-cropped.tif');
>> cpselect(f, g)
```

The tool helps navigate (zoom, pan, and scroll) in large images. Features (control points) can be selected and paired with each other by clicking on the images using the mouse.

Figure 6.18 shows the Control Point Selection Tool in action. Figure 6.18(a) is a binary image showing road, pond, stream, and power-line data. Figure 6.18(b) shows an aerial photograph covering the same region. The white rectangle in Fig. 6.18(b) shows the approximate location of the data in Fig. 6.18(a). Figure 6.18(c) is a screen shot of the Control Point Selection Tool showing six pairs of corresponding features selected at the intersections of several roadways.

6.7.3 Inferring Transformation Parameters Using `cp2tform`

Once feature pairs have been identified and matched, the next step in the image registration process is to determine the geometric transformation function. The usual procedure is to choose a particular transformation model and then estimate the necessary parameters. For example, one might determine that an affine transformation is appropriate and then use the corresponding feature pairs to derive the affine transform matrix.

The Image Processing Toolbox provides function `cp2tform` for inferring geometric transformation parameters from sets of feature pairs. The `cp2tform` syntax is:

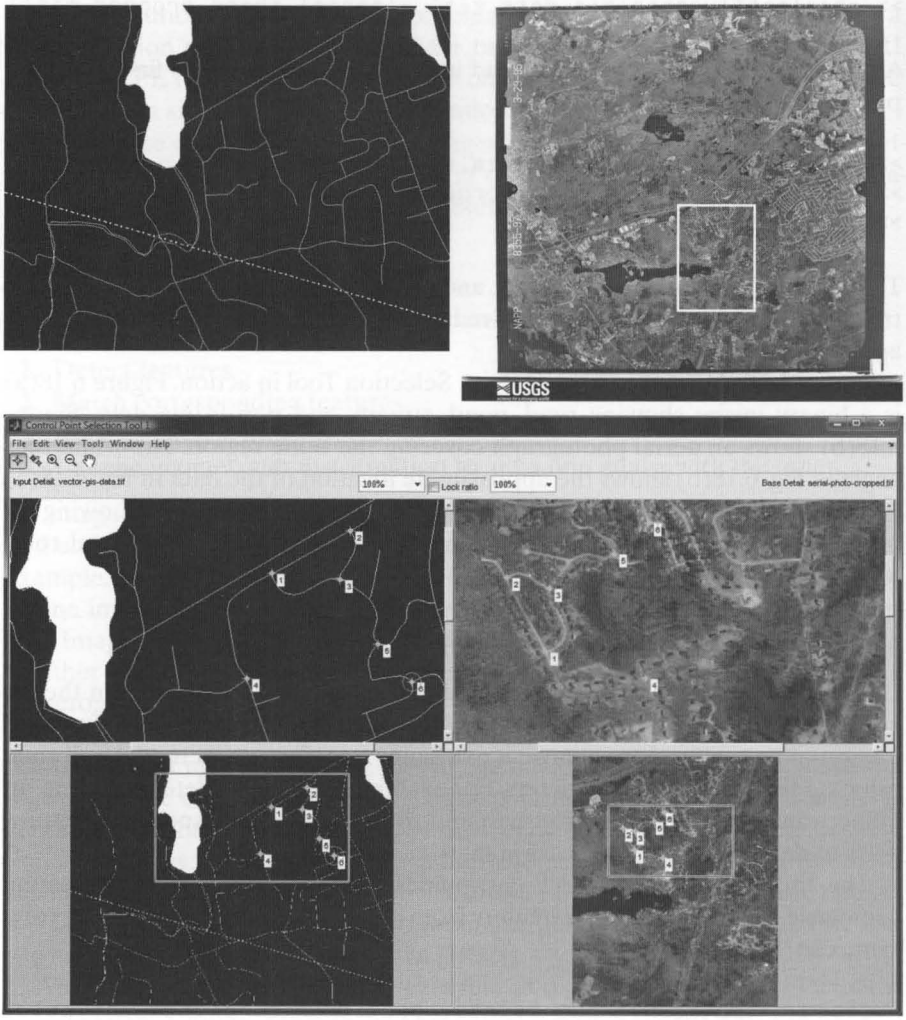
```
tform = cp2tform(input_points, base_points, transformtype)
```

The arguments `input_points` and `base_points` are both $P \times 2$ matrices containing corresponding feature locations. The third argument, `transformtype`, is a string (for example, `'affine'`) specifying the desired type of transformation. The output argument is a tform structure (see Section 6.1).

Table 6.2 lists all the different `tform` types that can be made with either `maketform` or `cp2tform`. The function `maketform` is used to specify transformation parameters directly, whereas `cp2tform` estimates transformation parameters using pairs of corresponding feature locations.

6.7.4 Visualizing Aligned Images

After a geometric transformation that aligns one image with another has been computed, the next step is often to visualize the two images together. One possible method is to display one image semi-transparently on top of the oth-



a b
c

FIGURE 6.18 Selecting and matching features using the Control Point Selection Tool (`cpselect`). (a) Binary image showing road and other data. (Original image courtesy of Office of Geographic and Environmental Information (MassGIS), Commonwealth of Massachusetts Executive Office of Environmental Affairs.) (b) Aerial photograph of the same region. (Original image courtesy of the USGS National Aerial Photography Program.) (c) Screen shot of the Control Point Selection Tool.

er. Several details have to be worked out because, even when registered, the images are likely to have different sizes and cover different regions of output space. Also, the output of the aligning geometric transformation is likely to include "out-of-bounds" pixels, usually displayed in black, as you have seen already. Out-of-bounds pixels from the transformed image should be displayed completely transparently so they do not obscure pixels in the other image. Custom function `visreg` listed below handles all these details automatically, making it easy to visualize two registered images.

`visreg`

```
function h = visreg(fref, f, tform, layer, alpha)
%VISREG Visualize registered images
%   VISREG(FREF, F, TFORM) displays two registered images together.
```

Type of Transformation	Description	Functions
Affine	Combination of scaling, rotation, shearing, and translation. Straight lines remain straight and parallel lines remain parallel.	`maketform` `cp2tform`
Box	Independent scaling and translation along each dimension; a subset of affine.	`maketform`
Composite	A collection of geometric transformations that are applied sequentially.	`maketform`
Custom	User-defined geometric transform; user provides functions that define $T\{\cdot\}$ and $T^{-1}\{\cdot\}$.	`maketform`
LWM	Local weighted mean; a locally-varying geometric transformation.	`cp2tform`
Nonreflective similarity	Combination of scaling, rotation, and translation. Straight lines remain straight, and parallel lines remain parallel. The basic shape of objects is preserved.	`cp2tform`
Piecewise linear	Locally varying geometric transformation. Different affine transformations are applied in triangular regions.	`cp2tform`
Polynomial	Geometric transformation in the form of a second-, third-, or fourth-order polynomial.	`cp2tform`
Projective	A superset of affine transformations. As with affine, straight lines remain straight, but parallel lines converge toward vanishing points.	`maketform` `cp2tform`
Similarity	Same as nonreflective similarity with the additional possibility of reflection.	`cp2tform`

TABLE 6.2 Transformation types supported by `cp2tform` and `maketform`.

```
%   FREF is the reference image. F is the input image, and TFORM
%   defines the geometric transformation that aligns image F with
%   image FREF.
%
%   VISREG(FREF, F, TFORM, LAYER) displays F transparently over FREF
%   if LAYER is 'top'; otherwise it displays FREF transparently over
%   F.
%
%   VISREG(FREF, F, TFORM, LAYER, ALPHA) uses the scalar value
%   ALPHA, which ranges between 0.0 and 1.0, to control the level of
%   transparency of the top image.  If ALPHA is 1.0, the top image
%   is opaque.  If ALPHA is 0.0, the top image is invisible.
%
%   H = VISREG(...) returns a vector of handles to the two displayed
%   image objects.  H is in the form [HBOTTOM, HTOP].
```

```
if nargin < 5
   alpha = 0.5;
end

if nargin < 4
   layer = 'top';
end

% Transform the input image, f, recording where the result lies in
% coordinate space.
[g, g_xdata, g_ydata] = imtransform(f, tform);

[M, N] = size(fref);
fref_xdata = [1 N];
fref_ydata = [1 M];

if strcmp(layer, 'top')
   % Display the transformed input image above the reference image.
   top_image = g;
   top_xdata = g_xdata;
   top_ydata = g_ydata;

   % The transformed input image is likely to have regions of black
   % pixels because they correspond to "out of bounds" locations on
   % the original image.  (See Example 6.2.)  These pixels should be
   % displayed completely transparently.  To compute the appropriate
   % transparency matrix, we can start with a matrix filled with the
   % value ALPHA and then transform it with the same transformation
   % applied to the input image.  Any zeros in the result will cause
   % the black "out of bounds" pixels in g to be displayed
   % transparently.
   top_alpha = imtransform(alpha * ones(size(f)), tform);

   bottom_image = fref;
   bottom_xdata = fref_xdata;
   bottom_ydata = fref_ydata;
else
   % Display the reference image above the transformed input image.
   top_image = fref;
   top_xdata = fref_xdata;
   top_ydata = fref_ydata;
   top_alpha = alpha;

   bottom_image = g;
   bottom_xdata = g_xdata;
   bottom_ydata = g_ydata;
end

% Display the bottom image at the correct location in coordinate
```

```
% space.
h_bottom = imshow(bottom_image, 'XData', bottom_xdata, ...
   'YData', bottom_ydata);
hold on

% Display the top image with the appropriate transparency.
h_top = imshow(top_image, 'XData', top_xdata, ...
   'YData', top_ydata);
set(h_top, 'AlphaData', top_alpha);

% The first call to imshow above has the effect of fixing the axis
% limits.  Use the axis command to let the axis limits be chosen
% automatically to fully encompass both images.
axis auto

if nargout > 0
   h = [h_bottom, h_top];
end
```

EXAMPLE 6.6: Visualizing registered images using `visreg`.

■ This example uses `cp2tform` and `visreg` to visualize the alignment of the images in Figs. 6.18(a) and (b). The matching feature pairs were selected manually, using the Control Point Selection Tool (`cpselect`), and saved to a MAT-file in a structure called `cpstruct`. Our first step is to load the images and `cpstruct`.

```
>> fref = imread('aerial-photo.tif');
>> f = imread('vector-gis-data.tif');
>> s = load('cpselect-results');
>> cpstruct = s.cpstruct;
```

The second step is to use `cp2tform` to infer an affine transformation that aligns image `f` with the reference image, `fref`.

```
>> tform = cp2tform(cpstruct, 'affine');
```

Third, we call `visreg` with the reference image, `fref`, the second image, `f`, and the geometric transformation that aligns `f` with `fref`. We accept the defaults for the fourth and fifth input arguments, so that the image `f` is displayed on top, with an `alpha` of 0.5 (meaning the pixels on top are one-half transparent).

```
>> visreg(fref, f, tform, axis([1740 2660 1710 2840]))
```

Figure 6.19 shows the result. ■

6.7.5 Area-Based Registration

An alternative to explicit feature selection and matching is *area-based registration*. In area-based registration, one image, called the *template image*, is shifted to cover each location in the second image. At each location, an area-based



FIGURE 6.19 Transparent overlay of registered images using `visreg`. (Note: the overlaid image was thickened using dilation to enhance its visibility. See Chapter 10 regarding dilation.)

similarity metric is computed. The template image is said to be a *match* at a particular position in the second image if a distinct peak in the similarity metric is found at that position.

One similarity metric used for area-based registration is *normalized cross-correlation* (also called the *correlation coefficient*). The definition of the normalized cross-correlation between an image and a template is:

See Section 13.3.3 for a more detailed discussion, and additional examples, of this function.

$$\gamma(x,y) = \frac{\sum_{s,t}\left[w(s,t) - \bar{w}\right]\left[f(x+s,y+t) - \bar{f}_{xy}\right]}{\sqrt{\sum_{s,t}\left[w(s,t) - \bar{w}\right]^2 \sum_{s,t}\left[f(x+s,y+t) - \bar{f}_{xy}\right]^2}}$$

where w is the template, $\bar{w}$ is the average value of the elements of the template (computed only once), f is the image, and $\bar{f}_{xy}$ is the average value of the image in the region where f and w overlap. The summation is taken over the values of s and t such that the image and the template overlap. The mechanics of computing the preceding expression for all values of x and y spanning the image are identical in principle to our discussion of correlation in Section 3.4.1. The main difference is in the actual computation performed at each pair of coordinates, (x, y). In this case, the purpose of the denominator is to normalize the metric with respect to variations in intensity. The value $\gamma(x, y)$ ranges from -1 to 1. A high value for $|\gamma(x, y)|$ indicates a good match between the template and the image, when the template is centered at coordinates (x, y).

The Image Processing Toolbox function for performing normalized cross-correlation is `normxcorr2`. Its calling syntax is:

```
g = normxcorr2(template, f)
```



EXAMPLE 6.7: Using function `normxcorr2` to locate a template in an image.

■ This example uses `normxcorr2` to find the location of the best match between a template and an image. First we read in the image and the template.

```
>> f = imread('car-left.jpg');
>> w = imread('car-template.jpg');
```

Figures 6.20(a) and (b) show the image and the template. Next we compute and display the normalized cross-correlation using `normxcorr2`.

```
>> g = normxcorr2(w, f);
>> imshow(\abs(g))
```

Figure 6.20(c) shows the normalized cross-correlation image (note the brightest spot, indicating a match between the template and the image). Now we search for the maximum value of `abs(g)` and determine its location. The location has to be adjusted for the size of the template, because the size of the output of `normxcorr2` is larger than the size the input image. (The size difference is the size of the template.)

```
>> gabs = abs(g);
>> [ypeak, xpeak] = find(gabs == max(gabs(:)));
>> ypeak = ypeak - (size(w, 1) - 1)/2;
>> xpeak = xpeak - (size(w, 2) - 1)/2;
>> imshow(f)
>> hold on
>> plot(xpeak, ypeak, 'wo')
```

Figure 6.20(d) shows the result. The small white circle indicates the center of the matched template area. ■

In addition to normalized cross-correlation, a number of other area-based similarity metrics have been proposed over the years in the image processing

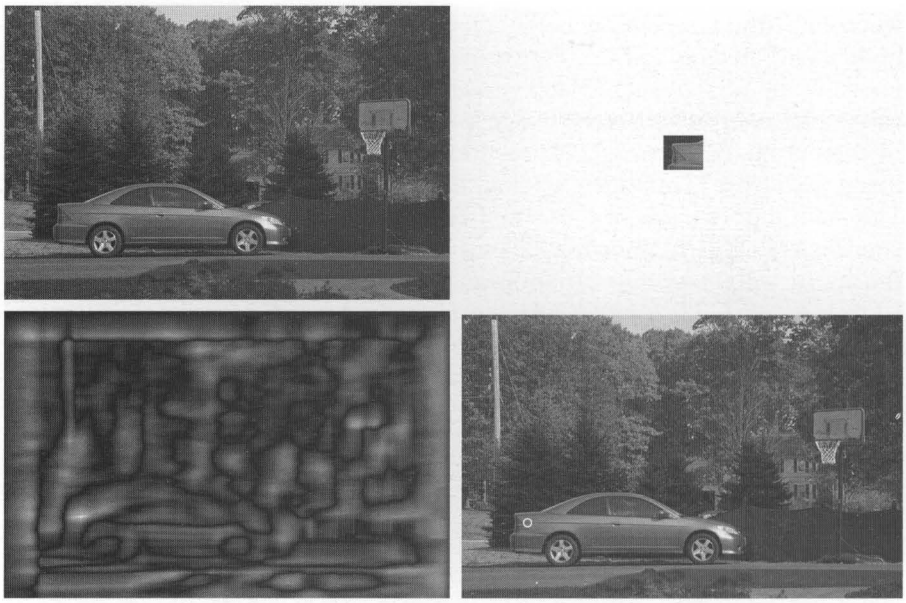


FIGURE 6.20
Using normalized cross-correlation to locate the best match between a template and an image. (a) Original image. (b) Template. (c) Absolute value of normalized cross-correlation. (d) Original image with small white circle indicating center of the matched template location.

literature, such as *sum of squared differences* and *sum of absolute differences*. The various metrics differ in factors such as computation time and robustness against outlier pixels (Brown [1992], Zitová [2003], and Szeliski [2006]).

In simple situations, template matching using normalized cross-correlation or other similarity metrics can be used to match up two overlapping images, such as those in Figs. 6.21(a) and (b). Given a template image contained in the area of overlap, the matched template locations in the two images can be compared, giving a translation vector that can be used to register the images. The next example illustrates this procedure.

EXAMPLE 6.8: Using `normxcorr2` to register two images differing by a translation.

■ This example uses `normxcorr2` and `visreg` to register the images in Figs. 6.21(a) and (b). First, read both images into the workspace:

```
>> f1 = imread('car-left.jpg');
>> f2 = imread('car-right.jpg');
```

The template image in Fig. 6.20(b) was cropped directly from one of the images and saved to a file.

```
>> w = imread('car-template.jpg');
```

Use `normxcorr2` to locate the template in both images.

```
>> g1 = normxcorr2(w, f1);
>> g2 = normxcorr2(w, f2);
```



a b
c

FIGURE 6.21
Using normalized cross-correlation to register overlapping images. (a) First image. (b) Second image. (c) Registered images as displayed using `visreg`.

Find the location of the maximum values of `g1` and `g2` and subtract the locations to determine the translation.

```
>> [y1, x1] = find(g1 == max(g1(:)));
>> [y2, x2] = find(g2 == max(g2(:)));
>> delta_x = x1 - x2

delta_x =

  -569

>> delta_y = y1 - y2

delta_y =

    -3
```

Once the relative translation between the images is found, we can form an affine tform structure and pass it to `visreg` to visualize the aligned images.

```
>> tform = maketform('affine', [1 0 0; 0 1 0; ...
    delta_x delta_y 1]);
>> visreg(f1, f2, tform)
```

Figure 6.21(c) shows the registered result. Although the images are well-aligned on the left portion of the overlap, they are slightly but visibly misaligned on the right. This is an indication that the geometric relationship between the two images is not completely characterized by a simple translation. ■

TABLE 6.3 Geometric transformation types for some image mosaicking scenarios (Goshtasby [2005], Brown [1992]).

Imaging Scenario	Geometric Transformation Type
Fixed camera location; horizontal optical axis; vertical axis of rotation through lens center; far scene.	Translation.
Fixed camera location; horizontal optical axis; vertical axis of rotation through lens center; close scene.	Map images onto cylinder, followed by translation.
Moving camera; same viewing angle; far scene.	Affine.
Moving camera; close, flat scene.	Projective.
Moving camera; close, nonflat scene.	Nonlinear, locally varying transformation; imaging geometry modeling may be necessary.

The process of registering overlapping images to produce a new image is called *image mosaicking*. Image mosaicking is often applied in remote sensing applications to build up a large-area view from smaller images, or in creating panoramic views. The mosaicking process involves determining geometric transformations that warp each of several images onto a common global coordinate system, and then blending overlapping pixels to make the result appear as seamless as possible. The type of geometric transformation chosen depends on the characteristics of the scene and the camera positions. Transformation types for a few common scenarios are described in Table 6.3. For more details about image mosaicking methods, see Goshtasby [2005] and Szeliski [2006].

6.7.5 Automatic Feature-Based Registration

The image registration methods discussed previously were partially manual processes. Example 6.6 relied upon manual selection and matching of feature points, while Example 6.8 used a manually chosen template. There are a variety of methods in use that are capable of fully automated image registration.

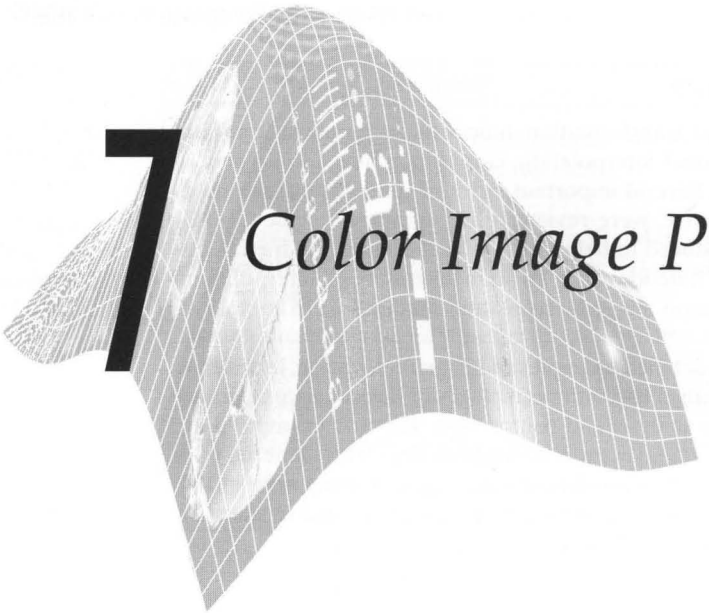
One widely used method involves using a *feature detector* to automatically choose a large number of potentially matchable feature points in both images. A commonly used feature detector is the Harris corner detector (see Section 12.3.5). The next step is to compute an initial set of possible matches using some feature-matching metric. Finally, an iterative technique known as RANSAC (random sampling and consensus) is applied (Fischler and Bolles [1981]).

Each RANSAC iteration selects a random subset of potential feature matches, from which a geometric transformation is derived. Feature matches that are consistent with the derived transformation are called *inliers*; inconsistent matches are called *outliers*. The iteration achieving the highest number of inliers is kept as the final solution. See Szeliski [2006] for detailed descriptions of this and many related methods.

Summary

This chapter explained how spatial transformation functions, in combination with inverse mapping and multidimensional interpolation, can be combined to achieve a variety of image processing effects. Several important types of spatial transformation functions, such as affine and projective, were reviewed and compared. A new MATLAB function, `vistform`, was introduced to help visualize and understand the effects of different spatial transformation functions. The basic mechanisms of interpolation were summarized, and several common image interpolation methods were compared in terms of speed and image quality.

The chapter concluded with two detailed examples of image registration, in which a geometric transformation is used to align two different images of the same scene, either for visualization or for quantitative analysis and comparison. The first example used manually selected control points to align vector road location data with an aerial photograph. The second example aligned two overlapping photographs using normalized cross correlation. A second visualization function, `visreg`, was introduced to transparently superimpose one aligned image over another.



7 Color Image Processing

Preview

In this chapter we discuss fundamentals of color image processing using the Image Processing Toolbox and extend some of its functionality by developing additional color generation and transformation functions. The discussion in this chapter assumes familiarity on the part of the reader with the principles and terminology of color image processing at an introductory level.

7.1 Color Image Representation in MATLAB

As noted in Section 2.6, the Image Processing Toolbox handles color images either as indexed images or RGB (red, green, blue) images. In this section we discuss these two image types in some detail.

7.1.1 RGB Images

An RGB *color image* is an $M \times N \times 3$ array of *color pixels*, where each color pixel is a triplet corresponding to the red, green, and blue components of an RGB image at a specific spatial location (see Fig. 7.1). An RGB image may be viewed as a "stack" of three gray-scale images that, when fed into the red, green, and blue inputs of a color monitor, produce a color image on the screen. By convention, the three images forming an RGB color image are referred to as the red, green, and blue *component images*. The data class of the component images determines their range of values. If an RGB image is of class `double`, the range of values is $[0, 1]$. Similarly, the range of values is $[0, 255]$ or $[0, 65535]$ for RGB images of class `uint8` or `uint16`, respectively. The number of bits used to represent the pixel values of the component images determines the *bit depth* of an RGB image. For example, if each component image is an 8-bit image, the corresponding RGB image is said to be 24 bits deep. Generally, the

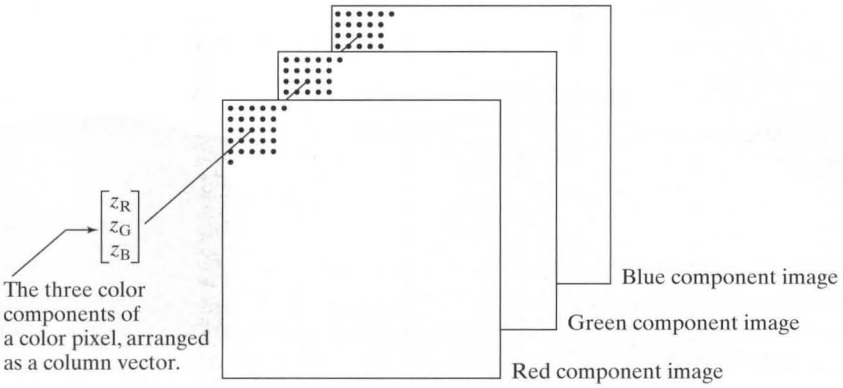
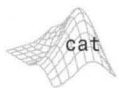


FIGURE 7.1 Schematic showing how pixels of an RGB color image are formed from the corresponding pixels of the three component images.

number of bits in all component images is the same. In this case, the number of possible colors in an RGB image is $(2^b)^3$ where b is the number of bits in each component image. For an 8-bit image, the number is 16,777,216 colors.

Let `fR`, `fG`, and `fB` represent three RGB component images. An RGB image is formed from these images by using the `cat` (concatenate) operator to stack the images:

```
rgb_image = cat(3, fR, fG, fB)
```



The order in which the images are placed in the operand matters. In general, `cat(dim, A1, A2, . . .)` concatenates the arrays (which must be of the same size) along the dimension specified by `dim`. For example, if `dim = 1`, the arrays are arranged vertically, if `dim = 2`, they are arranged horizontally, and, if `dim = 3`, they are stacked in the third dimension, as in Fig. 7.1.

If all component images are identical, the result is a gray-scale image. Let `rgb_image` denote an RGB image. The following commands extract the three component images:

```
>> fR = rgb_image(:, :, 1);
>> fG = rgb_image(:, :, 2);
>> fB = rgb_image(:, :, 3);
```

The RGB *color space* usually is shown graphically as an RGB color cube, as depicted in Fig. 7.2. The vertices of the cube are the *primary* (red, green, and blue) and *secondary* (cyan, magenta, and yellow) colors of light.

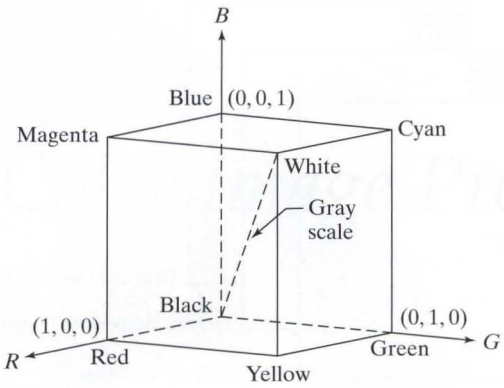
To view the color cube from any perspective, use custom function `rgbcube`:

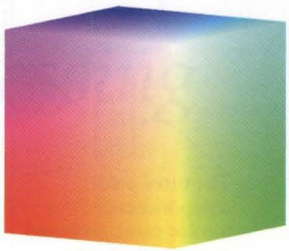
```
rgbcube(vx, vy, vz)
```

Typing `rgbcube(vx, vy, vz)` at the prompt produces an RGB cube on the MATLAB desktop, viewed from point `(vx, vy, vz)`. The resulting image can be saved to disk using function `print`, discussed in Section 2.4. The code for function `rgbcube` follows.

a b

FIGURE 7.2
(a) Schematic of the RGB color cube showing the primary and secondary colors of light at the vertices. Points along the main diagonal have gray values from black at the origin to white at point (1, 1, 1). (b) The RGB color cube.





rgbcube

```
function rgbcube(vx, vy, vz)
%RGBCUBE Displays an RGB cube on the MATLAB desktop.
%   RGBCUBE(VX, VY, VZ) displays an RGB color cube, viewed from point
%   (VX, VY, VZ). With no input arguments, RGBCUBE uses (10,10,4) as
%   the default viewing coordinates. To view individual color
%   planes, use the following viewing coordinates, where the first
%   color in the sequence is the closest to the viewing axis, and the
%   other colors are as seen from that axis, proceeding to the right
%   right (or above), and then moving clockwise.
%
%       ----------------------------------------------------
%            COLOR PLANE                    ( vx,  vy,  vz)
%       ----------------------------------------------------
%        Blue-Magenta-White-Cyan            (  0,   0,  10)
%        Red-Yellow-White-Magenta           ( 10,   0,   0)
%        Green-Cyan-White-Yellow            (  0,  10,   0)
%        Black-Red-Magenta-Blue             (  0, -10,   0)
%        Black-Blue-Cyan-Green              (-10,   0,   0)
%        Black-Red-Yellow-Green             (  0,   0, -10)

% Set up parameters for function patch.
vertices_matrix = [0 0 0;0 0 1;0 1 0;0 1 1;1 0 0;1 0 1;1 1 0;1 1 1];
faces_matrix = [1 5 6 2;1 3 7 5;1 2 4 3;2 4 8 6;3 7 8 4;5 6 8 7];
colors = vertices_matrix;
% The order of the cube vertices was selected to be the same as
% the  order of the (R,G,B) colors (e.g., (0,0,0) corresponds to
% black, (1, 1, 1) corresponds to white, and so on.)

% Generate RGB cube using function patch.
patch('Vertices', vertices_matrix, 'Faces', faces_matrix, ...
      'FaceVertexCData', colors, 'FaceColor', 'interp', ...
      'EdgeAlpha', 0)

% Set up viewing point.
if nargin == 0
   vx = 10; vy = 10; vz = 4;
```



patch

Function patch creates filled, 2-D polygons based on specified property/value pairs. For more information about patch, see the reference page for this function.

```
elseif nargin ~= 3
   error('Wrong number of inputs.')
end
axis off
view([vx, vy, vz])
axis square
```

7.1.2 Indexed Images

An *indexed* image has two components: a *data matrix* of integers, `X`, and a *color map matrix*, `map`. Matrix `map` is an $m \times 3$ array of class `double` containing floating-point values in the range [0, 1]. The length of the map is equal to the number of colors it defines. Each row of `map` specifies the red, green, and blue components of a single color (if the three columns of `map` are equal, the color map becomes a *gray-scale map*). An indexed image uses "direct mapping" of pixel intensity values to color-map values. The color of each pixel is determined by using the corresponding value of integer matrix `X` as an index (hence the name *indexed* image) into `map`. If `X` is of class `double`, then value 1 points to the first row in map, value 2 points to the second row, and so on. If `X` is of class `uint8` or `uint16`, then 0 points to the first row in `map`. These concepts are illustrated in Fig. 7.3.

To display an indexed image we write

```
>> imshow(X, map)
```

or, alternatively,

```
>> image(X)
>> colormap(map)
```

A color map is stored with an indexed image and is automatically loaded with the image when the `imread` function is used to load the image.

Sometimes it is necessary to approximate an indexed image by one with fewer colors. For this we use function `imapprox`, whose syntax is

```
[Y, newmap] = imapprox(X, map, n)
```



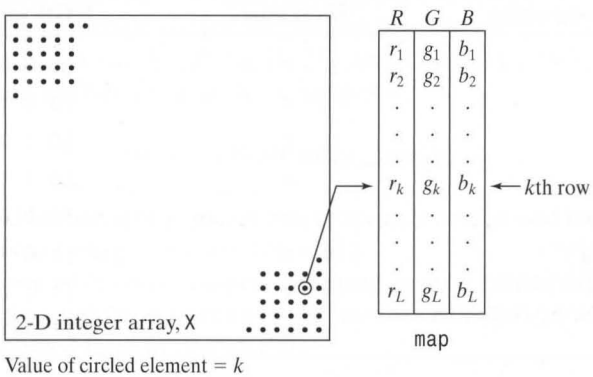


FIGURE 7.3 Elements of an indexed image. The value of an element of integer array `X` determines the row number in the color map. Each row contains an RGB triplet, and L is the total number of rows.

This function returns an array `Y` with color map `newmap`, which has at most `n` colors. The input array `X` can be of class `uint8`, `uint16`, or `double`. The output `Y` is of class `uint8` if `n` is less than or equal to 256. If `n` is greater than 256, `Y` is of class `double`.

When the number of rows in a map is less than the number of distinct integer values in `X`, multiple values in `X` are assigned the same color in the map. For example, suppose that `X` consists of four vertical bands of equal width, with values 1, 64, 128, and 256. If we specify the color map `map = [0 0 0; 1 1 1]`, then all the elements in `X` with value 1 would point to the first row (black) of the map and all the other elements would point to the second row (white). Thus, the command `imshow(X, map)` would display an image with a black band followed by three white bands. In fact, this would be true until the length of the map became 65, at which time the display would be a black band, followed by a gray band, followed by two white bands. Nonsensical image displays can result if the length of the map exceeds the allowed range of values of the elements of `X`.

There are several ways to specify a color map. One approach is to use the statement

```
>> map(k, :) = [r(k) g(k) b(k)];
```

where `[r(k) g(k) b(k)]` are RGB values that specify one row of a color map. The map is filled out by varying `k`.

Table 7.1 lists the RGB values of several basic colors. Any of the three formats shown in the table can be used to specify colors. For example, the background color of a figure can be changed to green by using any of the following three statements:




```
>> whitebg('g');
>> whitebg('green');
>> whitebg([0 1 0]);
```

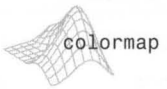
Other colors in addition to the ones in Table 7.1 involve fractional values. For instance, `[.5 .5 .5]` is gray, `[.5 0 0]` is dark red, and `[.49 1 .83]` is aquamarine.

TABLE 7.1
RGB values of some basic colors. The long or short names (enclosed by single quotes) can be used instead of a numerical triplet to specify an RGB color.

Long name	Short name	RGB values
Black	k	[0 0 0]
Blue	b	[0 0 1]
Green	g	[0 1 0]
Cyan	c	[0 1 1]
Red	r	[1 0 0]
Magenta	m	[1 0 1]
Yellow	y	[1 1 0]
White	w	[1 1 1]

MATLAB provides several predefined color maps, accessed using the command

```
>> colormap(map_name);
```



which sets the color map to the matrix `map_name`; an example is

```
>> colormap(copper)
```



where `copper` is a MATLAB color map function. The colors in this mapping vary smoothly from black to bright copper. If the last image displayed was an indexed image, this command changes its color map to `copper`. Alternatively, the image can be displayed directly with the desired color map:

```
>> imshow(X, copper)
```

Table 7.2 lists the predefined color maps available in MATLAB. The length (number of colors) of these color maps can be specified by enclosing the number in parentheses. For example, `gray(8)` generates a color map with 8 shades of gray.

7.1.3 Functions for Manipulating RGB and Indexed Images

Table 7.3 lists the toolbox functions suitable for converting between RGB, indexed, and gray-scale images. For clarity of notation in this section, we use `rgb_image` to denote RGB images, `gray_image` to denote gray-scale images, `bw` to denote black and white (binary) images, and `X`, to denote the data matrix component of indexed images. Recall that an indexed image is composed of an integer data matrix and a color map matrix.

Function `dither` applies both to gray-scale and to color images. Dithering is a process used routinely in the printing and publishing industry to give the visual impression of shade variations on a printed page that consists of dots. In the case of gray-scale images, dithering attempts to capture shades of gray by producing a binary image of black dots on a white background (or vice versa). The sizes of the dots vary, from small dots in light areas to increasingly larger dots for dark areas. The key issue in implementing a dithering algorithm is a trade off between "accuracy" of visual perception and computational complexity. The dithering approach used in the toolbox is based on the Floyd-Steinberg algorithm (see Floyd and Steinberg [1975], and Ulichney [1987]). The syntax used by function `dither` for gray-scale images is

```
bw = dither(gray_image)
```



where, as noted earlier, `gray_image` is a gray-scale image and `bw` is the resulting dithered binary image (of class `logical`).

When working with color images, dithering is used principally in conjunction with function `rgb2ind` to reduce the number of colors in an image. This function is discussed later in this section.

TABLE 7.2 MATLAB predefined color maps.

Function	Description
`autumn`	Varies smoothly from red, through orange, to yellow.
`bone`	A gray-scale color map with a higher value for the blue component. This color map is useful for adding an "electronic" look to gray-scale images.
`colorcube`	Contains as many regularly spaced colors in RGB color space as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
`cool`	Consists of colors that are smoothly-varying shades from cyan to magenta.
`copper`	Varies smoothly from black to bright copper.
`flag`	Consists of the colors red, white, blue, and black. This color map completely changes color with each index increment.
`gray`	Returns a linear gray-scale color map.
`hot`	Varies smoothly from black, through shades of red, orange, and yellow, to white.
`hsv`	Varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The color map is particularly appropriate for displaying periodic functions.
`jet`	Ranges from blue to red, and passes through the colors cyan, yellow, and orange.
`lines`	Produces a color map of colors specified by the axes `ColorOrder` property and a shade of gray. Consult the help page for function `ColorOrder` for details on this function.
`pink`	Contains pastel shades of pink. The pink color map provides sepia tone colorization of gray-scale photographs.
`prism`	Repeats the six colors red, orange, yellow, green, blue, and violet.
`spring`	Consists of colors that are shades of magenta and yellow.
`summer`	Consists of colors that are shades of green and yellow.
`winter`	Consists of colors that are shades of blue and green.
`white`	This is an all white monochrome color map.

TABLE 7.3 Toolbox functions for converting between RGB, indexed, and gray-scale images.

Function	Description
`dither`	Creates an indexed image from an RGB image by dithering.
`grayslice`	Creates an indexed image from a gray-scale intensity image by thresholding.
`gray2ind`	Creates and indexed image from a gray-scale intensity image.
`ind2gray`	Creates a gray-scale image from an indexed image.
`rgb2ind`	Creates an indexed image from an RGB image.
`ind2rgb`	Creates an RGB image from an indexed image.
`rgb2gray`	Creates a gray-scale image from an RGB image.

Function `grayslice` has the syntax

```
X = grayslice(gray_image, n)
```



This function produces an indexed image by thresholding a gray-scale image with threshold values

$$\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$$

As noted earlier, the resulting indexed image can be viewed with the command `imshow(X, map)` using a map of appropriate length [e.g., `jet(16)`]. An alternate syntax is

```
X = grayslice(gray_image, v)
```

where `v` is a vector (with values in the range [0, 1]) used to threshold `gray_image`. Function `grayslice` is a basic tool for pseudocolor image processing, where specified gray intensity bands are assigned different colors. The input image can be of class `uint8`, `uint16`, or `double`. The threshold values in `v` must be in the range [0, 1], even if the input image is of class `uint8` or `uint16`. The function performs the necessary scaling.

Function `gray2ind`, with syntax

```
[X, map] = gray2ind(gray_image, n)
```



scales, then rounds image `gray_image` to produce an indexed image `X` with color map `gray(n)`. If `n` is omitted, it defaults to 64. The input image can be of class `uint8`, `uint16`, or `double`. The class of the output image `X` is `uint8` if `n` is less than or equal to 256, or of class `uint16` if `n` is greater than 256.

Function `ind2gray`, with syntax

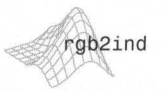
```
gray_image = ind2gray(X, map)
```



converts an indexed image, composed of `X` and `map`, to a gray-scale image. Array `X` can be of class `uint8`, `uint16`, or `double`. The output image is of class `double`.

The syntax of interest in this chapter for function `rgb2ind` has the form

```
[X, map] = rgb2ind(rgb_image, n, dither_option)
```



where `n` determines the number of colors of `map`, and `dither_option` can have one of two values: `'dither'` (the default) dithers, if necessary, to achieve better color resolution at the expense of spatial resolution; conversely, `'nodither'` maps each color in the original image to the closest color in the new map (depending on the value of `n`); no dithering is performed. The input image can be of class `uint8`, `uint16`, or `double`. The output array, `X`, is of class `uint8` if `n` is less

than or equal to 256; otherwise it is of class `uint16`. Example 7.1 shows the effect that dithering has on color reduction.

Function `ind2rgb`, with syntax



```
rgb_image = ind2rgb(X, map)
```

converts the matrix `X` and corresponding color map `map` to RGB format; `X` can be of class `uint8`, `uint16`, or `double`. The output RGB image is an $M \times N \times 3$ array of class `double`.

Finally, function `rgb2gray`, with syntax



```
gray_image = rgb2gray(rgb_image)
```

converts an RGB image to a gray-scale image. The input RGB image can be of class `uint8`, `uint16`, or `double`; the output image is of the same class as the input.

EXAMPLE 7.1: Illustration of some of the functions in Table 7.3.

■ Function `rgb2ind` is useful for reducing the number of colors in an RGB image. As an illustration of this function, and of the advantages of using the dithering option, consider Fig. 7.4(a), which is a 24-bit RGB image, `f`. Figures 7.4(b) and (c) show the results of using the commands

```
>> [X1, map1] = rgb2ind(f, 8, 'nodither');
>> imshow(X1, map1)
```

and

```
>> [X2, map2] = rgb2ind(f, 8, 'dither');
>> figure, imshow(X2, map2)
```

Both images have only 8 colors, which is a significant reduction in the 16 million possible colors of `uint8` image `f`. Figure 7.4(b) shows a very noticeable degree of false contouring, especially in the center of the large flower. The dithered image shows better tonality, and considerably less false contouring, a result of the "randomness" introduced by dithering. The image is a little blurred, but it certainly is visually superior to Fig. 7.4(b).

The effects of dithering usually are better illustrated with a grayscale image. Figures 7.4(d) and (e) were obtained using the commands

```
>> g = rgb2gray(f);
>> g1 = dither(g);
>> figure, imshow(g); figure, imshow(g1)
```

The image in Fig. 7.4(e) is binary, which again represents a significant degree of data reduction. Figures. 7.4(c) and (e) demonstrate why dithering is such a staple in the printing and publishing industry, especially in situations (such as in newspapers) in which paper quality and printing resolution are low. ■



a
b c
d e

FIGURE 7.4
(a) RGB image. (b) Number of colors reduced to 8, with no dithering. (c) Number of colors reduced to 8, with dithering. (d) Gray-scale version of (a) obtained using function `rgb2gray`. (e) Dithered gray-scale image (this is a binary image).

7.2 Converting Between Color Spaces

As explained in the previous section, the toolbox represents colors as RGB values, directly in an RGB image, or indirectly in an indexed image, where the color map is stored in RGB format. However, there are other color spaces (also called *color models*) whose use in some applications may be more convenient and/or meaningful than RGB. These models are transformations of the RGB model and include the NTSC, YCbCr, HSV, CMY, CMYK, and HSI color spaces. The toolbox provides conversion functions from RGB to the NTSC, YCbCr, HSV and CMY color spaces, and back. Custom functions for converting to and from the HSI color space are developed later in this section.

7.2.1 NTSC Color Space

The NTSC color system is used in analog television. One of the main advantages of this format is that gray-scale information is separate from color data, so the same signal can be used for both color and monochrome television sets. In the NTSC format, image data consists of three components: *luminance* (Y), *hue* (I), and *saturation* (Q), where the choice of the letters YIQ is conventional. The luminance component represents gray-scale information, and the other two components carry the color information of a TV signal. The YIQ components are obtained from the RGB components of an image using the linear transformation

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Note that the elements of the first row sum to 1 and the elements of the next two rows sum to 0. This is as expected because for a grayscale image all the RGB components are equal, so the I and Q components should be 0 for such an image. Function `rgb2ntsc` performs the preceding transformation:




```
yiq_image = rgb2ntsc(rgb_image)
```

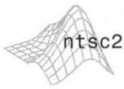
where the input RGB image can be of class `uint8`, `uint16`, or `double`. The output image is an $M \times N \times 3$ array of class `double`. Component image `yiq_image(:, :, 1)` is the luminance, `yiq_image(:, :, 2)` is the hue, and `yiq_image(:, :, 3)` is the saturation image.

Similarly, the RGB components are obtained from the YIQ components using the linear transformation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Toolbox function `ntsc2rgb` implements this transformation. The syntax is

```
rgb_image = ntsc2rgb(yiq_image)
```



Both the input and output images are of class `double`.

7.2.2 The YCbCr Color Space

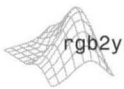
The YCbCr color space is used extensively in digital video. In this format, luminance information is represented by a single component, Y, and color information is stored as two color-difference components, Cb and Cr. Component Cb is the difference between the blue component and a reference value, and component Cr is the difference between the red component and a reference value (Poynton [1996]). The transformation used by the toolbox to convert from RGB to YCbCr is

To see the transformation matrix used to convert from YCbCr to RGB, type the following command at the prompt:
`>> edit ycbcr2rgb`

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.000 \\ 112.000 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

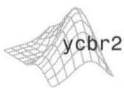
The conversion function is

```
ycbcr_image = rgb2ycbcr(rgb_image)
```



The input RGB image can be of class `uint8`, `uint16`, or `double`. The output image is of the same class as the input. A similar transformation converts from YCbCr back to RGB:

```
rgb_image = ycbr2rgb(ycbcr_image)
```



The input YCbCr image can be of class `uint8`, `uint16`, or `double`. The output image is of the same class as the input.

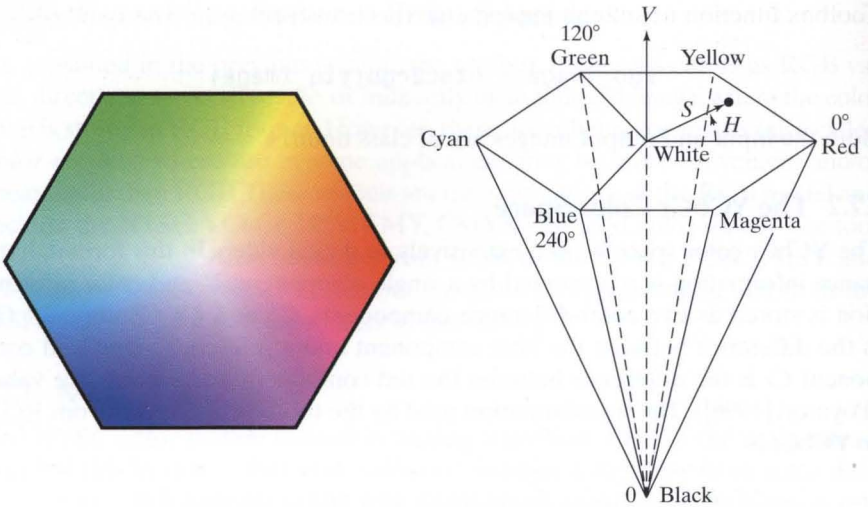
7.2.3 The HSV Color Space

HSV (hue, saturation, value) is one of several color systems used by people to select colors (e.g., of paints or inks) from a color wheel or palette. This color system is considerably closer than the RGB system to the way in which humans experience and describe color sensations. In artists' terminology, hue, saturation, and value refer approximately to tint, shade, and tone.

The HSV color space is formulated by looking at the RGB color cube along its gray axis (the axis joining the black and white vertices), which results in the hexagonally shaped color palette shown in Fig. 7.5(a). As we move along the vertical (gray) axis in Fig. 7.5(b), the size of the hexagonal plane that is perpendicular to the axis changes, yielding the volume depicted in the figure. Hue is expressed as an angle around a color hexagon, typically using the red axis as the reference (0°) axis. The value component is measured along the axis of the cone.

a b

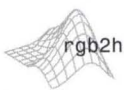
FIGURE 7.5
(a) The HSV color hexagon. (b) The HSV hexagonal cone.



The $V = 0$ end of the axis is black. The $V = 1$ end of the axis is white, which lies in the center of the full color hexagon in Fig. 7.5(a). Thus, this axis represents all shades of gray. Saturation (purity of the color) is measured as the distance from the V axis.

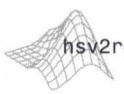
The HSV color system is based on cylindrical coordinates. Converting from RGB to HSV entails developing the equations to map RGB values (which are in Cartesian coordinates) to cylindrical coordinates. This topic is treated in detail in most texts on computer graphics (e.g., see Rogers [1997]) so we do not develop the equations here.

The MATLAB function for converting from RGB to HSV is `rgb2hsv`, whose syntax is




```
hsv_image = rgb2hsv(rgb_image)
```

The input RGB image can be of class `uint8`, `uint16`, or `double`; the output image is of class `double`. The function for converting from HSV back to RGB is `hsv2rgb`:




```
rgb_image = hsv2rgb(hsv_image)
```

The input image must be of class `double`. The output is of class `double` also.

7.2.4 The CMY and CMYK Color Spaces

Cyan, magenta, and yellow are the secondary colors of light or, alternatively, the primary colors of pigments. For example, when a surface coated with cyan pigment is illuminated with white light, no red light is reflected from the surface. That is, the cyan pigment subtracts red light from the light reflected by the surface.

Most devices that deposit colored pigments on paper, such as color printers and copiers, require CMY data input or perform an RGB to CMY conversion internally. An approximate conversion can be performed using the equation

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

where the assumption is that all color values have been normalized to the range [0, 1]. This equation demonstrates the statement in the previous paragraph that light reflected from a surface coated with pure cyan does not contain red (that is, $C = 1 - R$ in the equation). Similarly, pure magenta does not reflect green, and pure yellow does not reflect blue. The preceding equation also shows that RGB values can be obtained easily from a set of CMY values by subtracting the individual CMY values from 1.

In theory, equal amounts of the pigment primaries, cyan, magenta, and yellow should produce black. In practice, combining these colors for printing produces a muddy-looking black. So, in order to produce true black (which is the predominant color in printing), a fourth color, *black*, is added, giving rise to the CMYK color model. Thus, when publishers talk about "four-color printing," they are referring to the three-colors of the CMY color model plus black.

Function `imcomplement` introduced in Section 3.2.1 can be used to perform the approximate conversion from RGB to CMY:

```
cmy_image = imcomplement(rgb_image)
```

We use this function also to convert a CMY image to RGB:

```
rgb_image = imcomplement(cmy_image)
```

A high-quality conversion to CMY or CMYK requires specific knowledge of printer inks and media, as well as heuristic methods for determining where to use black ink (K) instead of the other three inks. This conversion can be accomplished using an *ICC color profile* created for a particular printer (see Section 7.2.6 regarding ICC profiles).

7.2.5 The HSI Color Space

With the exception of HSV, the color spaces discussed thus far are not well suited for *describing* colors in terms that are practical for human interpretation. For example, one does not refer to the color of an automobile by giving the percentage of each of the pigment primaries composing its color.

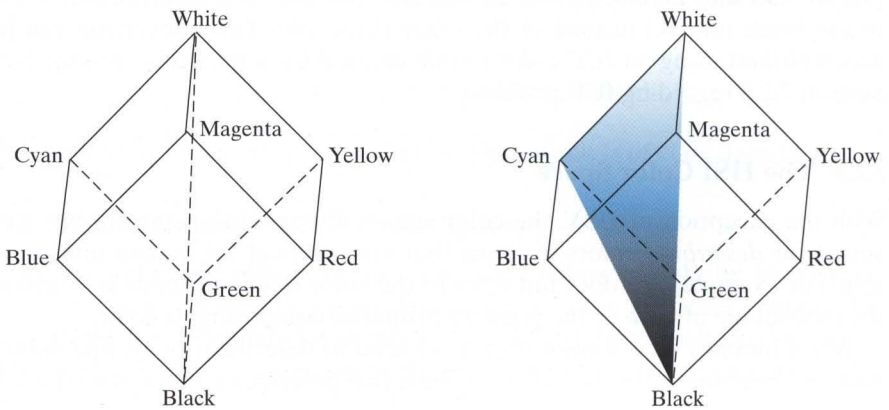
When humans view a color object, we tend to describe it by its hue, saturation, and brightness. Hue is a color attribute that describes a pure color, whereas

saturation gives a measure of the degree to which a pure color is diluted by white light. Brightness is a subjective descriptor that is practically impossible to measure. It embodies the achromatic notion of *intensity* and is one of the key factors in describing color sensation. We do know that intensity (gray level) is a most useful descriptor of monochromatic images. This quantity definitely is measurable and easily interpretable.

The color space we are about to present, called the *HSI* (hue, saturation, intensity) *color space*, decouples the intensity component from the color-carrying information (hue and saturation) in a color image. As a result, the HSI model is an ideal tool for developing image-processing algorithms based on color descriptions that are natural and intuitive to humans who, after all, are the developers and users of these algorithms. The HSV color space is somewhat similar, but its focus is on presenting colors that are meaningful when interpreted in terms of an artist's color palette.

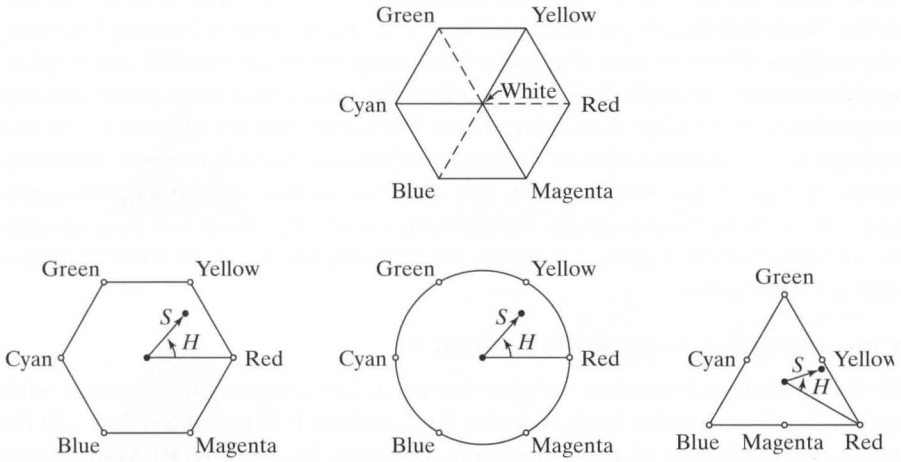
As discussed in Section 7.1.1, an RGB color image is composed of three monochrome intensity images, so it should come as no surprise that we should be able to extract intensity from an RGB image. This becomes evident if we take the color cube from Fig. 7.2 and stand it on the black, (0, 0, 0), vertex, with the white vertex, (1, 1, 1), directly above it, as in Fig. 7.6(a). As noted in connection with Fig. 7.2, the intensity is along the line joining these two vertices. In the arrangement shown in Fig. 7.6, the line (intensity axis) joining the black and white vertices is vertical. Thus, if we wanted to determine the intensity component of any color point in Fig. 7.6, we would simply pass a plane *perpendicular* to the intensity axis and containing the color point. The intersection of the plane with the intensity axis would give us an intensity value in the range [0, 1]. We also note with a little thought that the saturation (purity) of a color increases as a function of distance from the intensity axis. In fact, the saturation of points on the intensity axis is zero, as evidenced by the fact that all points along this axis are shades of gray.

In order to see how hue can be determined from a given RGB point, consider Fig. 7.6(b), which shows a plane defined by three points, (black, white,



a b

FIGURE 7.6 Relationship between the RGB and HSI color models.



a
b c d

FIGURE 7.7
Hue and saturation in the HSI color model. The dot is an arbitrary color point. The angle from the red axis gives the hue, and the length of the vector is the saturation. The intensity of all colors in any of these planes is given by the position of the plane on the vertical intensity axis.

and cyan). The fact that the black and white points are contained in the plane tells us that the intensity axis also is contained in that plane. Furthermore, we see that *all* points contained in the plane segment defined by the intensity axis and the boundaries of the cube have the *same* hue (cyan in this case). This is because the colors inside a color triangle are various combinations or mixtures of the three vertex colors. If two of those vertices are black and white, and the third is a color point, all points on the triangle must have the same hue because the black and white components do not contribute to changes in hue (of course, the intensity and saturation of points in this triangle do change). By rotating the shaded plane about the vertical intensity axis, we would obtain different hues. We conclude from these concepts that the hue, saturation, and intensity values required to form the HSI space can be obtained from the RGB color cube. That is, we can convert any RGB point to a corresponding point is the HSI color model by working out the geometrical formulas describing the reasoning just outlined.

Based on the preceding discussion, we see that the HSI space consists of a vertical intensity axis and the locus of color points that lie on a plane perpendicular to this axis. As the plane moves up and down the intensity axis, the boundaries defined by the intersection of the plane with the faces of the cube have either a triangular or hexagonal shape. This can be visualized much more readily by looking at the cube down its gray-scale axis, as in Fig. 7.7(a). In this plane we see that the primary colors are separated by 120°. The secondary colors are 60° from the primaries, which means that the angle between secondary colors is 120° also.

Figure 7.7(b) shows the hexagonal shape and an arbitrary color point (shown as a dot). The hue of the point is determined by an angle from some reference point. Usually (but not always) an angle of 0° from the red axis designates 0

hue, and the hue increases counterclockwise from there. The saturation (distance from the vertical axis) is the length of the vector from the origin to the point. Note that the origin is defined by the intersection of the color plane with the vertical intensity axis. The important components of the HSI color space are the vertical intensity axis, the length of the vector to a color point, and the angle this vector makes with the red axis. Therefore, it is not unusual to see the HSI planes defined is terms of the hexagon just discussed, a triangle, or even a circle, as Figs. 7.7(c) and (d) show. The shape chosen is not important because any one of these shapes can be warped into one of the others two by a geometric transformation. Figure 7.8 shows the HSI model based on color triangles and also on circles.

Converting Colors from RGB to HSI

In the following discussion we give the necessary conversion equations without derivation. See the book web site (the address is listed in Section 1.5) for a detailed derivation of these equations. Given an image in RGB color format, the H component of each RGB pixel is obtained using the equation

$$H = \begin{cases} \theta & \text{if } B \leq G \\ 360 - \theta & \text{if } B > G \end{cases}$$

with

$$\theta = \cos^{-1}\left\{\frac{0.5[(R-G)+(R-B)]}{\left[(R-G)^2+(R-B)(G-B)\right]^{1/2}}\right\}$$

The saturation component is given by

$$S = 1 - \frac{3}{(R+G+B)}[\min(R,G,B)]$$

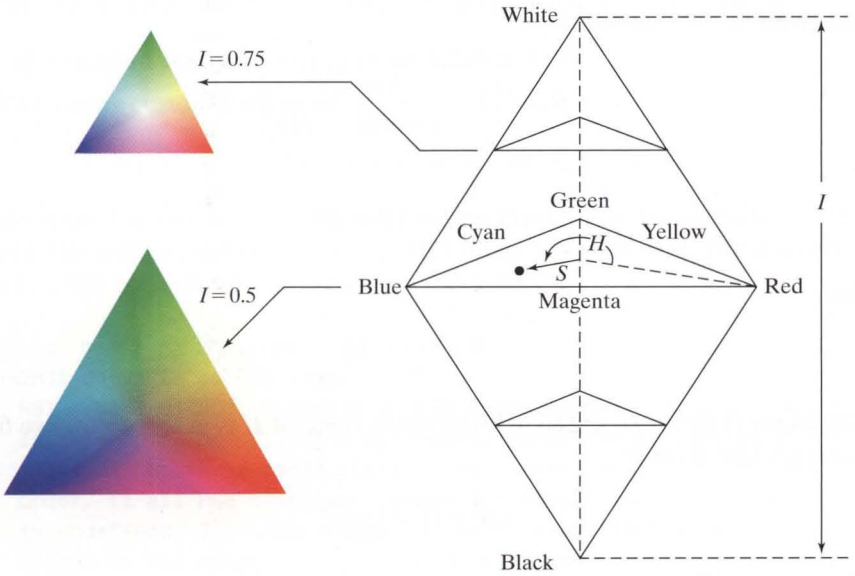
Finally, the intensity component is given by

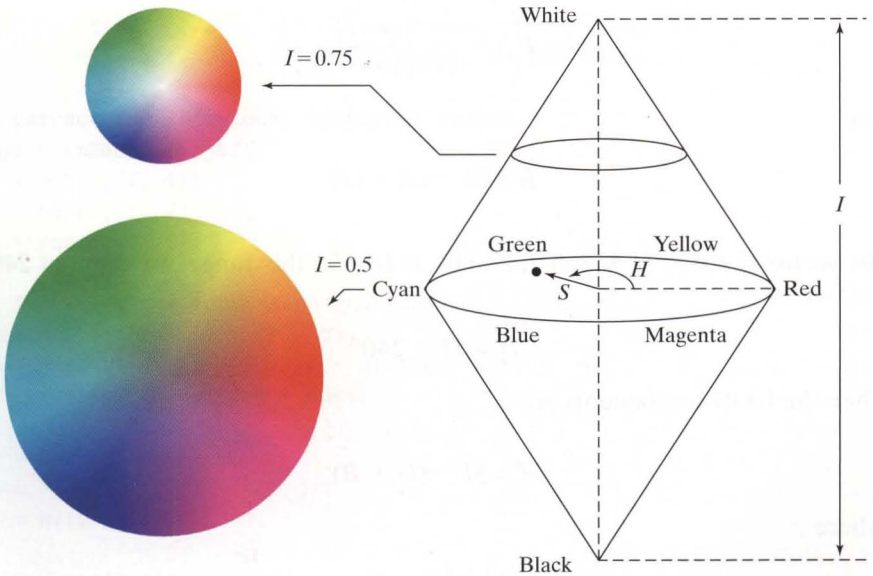
$$I = \frac{1}{3}(R+G+B)$$

It is assumed that the RGB values have been normalized to the range $[0, 1]$, and that angle θ is measured with respect to the red axis of the HSI space, as indicated in Fig. 7.7. Hue can be normalized to the range $[0, 1]$ by dividing by $360°$ all values resulting from the equation for H. The other two HSI components already are in this range if the given RGB values are in the interval $[0, 1]$.

Converting Colors from HSI to RGB

Given values of HSI in the interval $[0, 1]$, we now wish to find the corresponding RGB values in the same range. The applicable equations depend on the values of H. There are three sectors of interest, corresponding to the $120°$





a
b

FIGURE 7.8
The HSI color model based on (a) triangular and (b) circular color planes. The triangles and circles are perpendicular to the vertical intensity axis.

intervals between the primaries, as mentioned earlier. We begin by multiplying H by 360°, which returns the hue to its original range of [0°, 360°].

RG sector ($0° \leq H < 120°$): When H is in this sector, the RGB components are given by the equations

$$R = I\left[1 + \frac{S\cos H}{\cos(60° - H)}\right]$$

$$G = 3I - (R + B)$$

and

$$B = I(1 - S)$$

GB sector ($120° \leq H < 240°$): If the given value of H is in this sector, we first subtract 120° from it:

$$H = H - 120°$$

Then the RGB components are

$$R = I(1 - S)$$

$$G = I\left[1 + \frac{S\cos H}{\cos(60° - H)}\right]$$

and

$$B = 3I - (R + G)$$

BR sector ($240° \leq H \leq 360°$): Finally, if H is in this range, we subtract 240° from it:

$$H = H - 240°$$

Then the RGB components are

$$R = 3I - (G + B)$$

where

$$G = I(1 - S)$$

and

$$B = I\left[1 + \frac{S\cos H}{\cos(60^\circ - H)}\right]$$

We show how to use these equations for image processing in Section 7.5.1.

An M-function for Converting from RGB to HSI

The following custom function,

```
hsi = rgb2hsi(rgb)
```

implements the equations just discussed for converting from RGB to HSI, where `rgb` and `hsi` denote RGB and HSI images, respectively. The documentation in the code details the use of this function.

rgb2hsi

```
function hsi = rgb2hsi(rgb)
%RGB2HSI Converts an RGB image to HSI.
%   HSI = RGB2HSI(RGB) converts an RGB image to HSI. The input image
%   is assumed to be of size M-by-N-by-3, where the third dimension
%   accounts for three image planes: red, green, and blue, in that
%   order. If all RGB component images are equal, the HSI conversion
%   is undefined. The input image can be of class double (with
%   values in the range [0, 1]), uint8, or uint16.
%
%   The output image, HSI, is of class double, where:
%      HSI(:, :, 1) = hue image normalized to the range [0,1] by
%                     dividing all angle values by 2*pi.
%      HSI(:, :, 2) = saturation image, in the range [0, 1].
%      HSI(:, :, 3) = intensity image, in the range [0, 1].

% Extract the individual component images.
rgb = im2double(rgb);
r = rgb(:, :, 1);
g = rgb(:, :, 2);
b = rgb(:, :, 3);

% Implement the conversion equations.
num = 0.5*((r - g) + (r - b));
den = sqrt((r - g).^2 + (r - b).*(g - b));
theta = acos(num./(den + eps));

H = theta;
H(b > g) = 2*pi - H(b > g);
H = H/(2*pi);

num = min(min(r, g), b);
den = r + g + b;
den(den == 0) = eps;
S = 1 - 3.* num./den;
```

```
H(S == 0) = 0;
I = (r + g + b)/3;

% Combine all three results into an hsi image.
hsi = cat(3, H, S, I);
```

An M-function for Converting from HSI to RGB

The following function,

```
rgb = hsi2rgb(hsi)
```

implements the equations for converting from HSI to RGB. The documentation in the code details the use of this function.

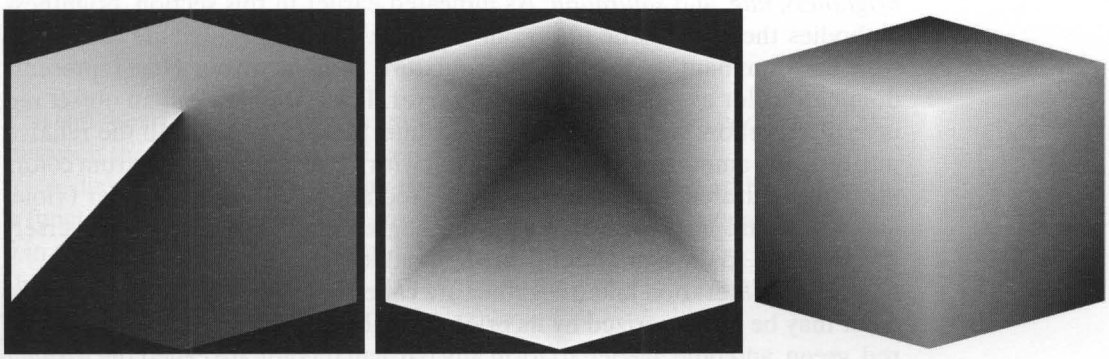
hsi2rgb

```
function rgb = hsi2rgb(hsi)
%HSI2RGB Converts an HSI image to RGB.
%   RGB = HSI2RGB(HSI) converts an HSI image RGB, where HSI is
%   assumed to be of class double with:
%      HSI(:, :, 1) = hue image, assumed to be in the range
%                     [0, 1] by having been divided by 2*pi.
%      HSI(:, :, 2) = saturation image, in the range [0, 1];
%      HSI(:, :, 3) = intensity image, in the range [0, 1].
%
%   The components of the output image are:
%      RGB(:, :, 1) = red.
%      RGB(:, :, 2) = green.
%      RGB(:, :, 3) = blue.

% Extract the individual HSI component images.
H = hsi(:, :, 1) * 2 * pi;
S = hsi(:, :, 2);
I = hsi(:, :, 3);

% Implement the conversion equations.
R = zeros(size(hsi, 1), size(hsi, 2));
G = zeros(size(hsi, 1), size(hsi, 2));
B = zeros(size(hsi, 1), size(hsi, 2));

% RG sector (0 <= H < 2*pi/3).
idx = find( (0 <= H) & (H < 2*pi/3));
B(idx) = I(idx) .* (1 - S(idx));
R(idx) = I(idx) .* (1 + S(idx) .* cos(H(idx))./ ...
                                        cos(pi/3 - H(idx)));
G(idx) = 3*I(idx) - (R(idx) + B(idx));

% BG sector (2*pi/3 <= H < 4*pi/3).
idx = find( (2*pi/3 <= H) & (H < 4*pi/3) );
```

```
R(idx) = I(idx) .* (1 - S(idx));
G(idx) = I(idx) .* (1 + S(idx) .* cos(H(idx) - 2*pi/3) ./ ...
                    cos (pi - H(idx)));
B(idx) = 3*I(idx) - (R(idx) + G(idx));

% BR sector.
idx = find( (4*pi/3 <= H) & (H <= 2*pi));
G(idx) = I(idx).* (1 - S(idx));
B(idx) = I(idx).* (1 + S(idx).* cos(H(idx) - 4*pi/3)./ ...
                                    cos(5*pi/3 - H(idx)));
R(idx) = 3*I(idx) - (G(idx) + B(idx));

% Combine all three results into an RGB image. Clip to [0, 1] to
% compensate for floating-point arithmetic rounding effects.
rgb = cat(3, R, G, B);
rgb = max(min(rgb, 1), 0);
```

EXAMPLE 7.2: Converting from RGB to HSI.

■ Figure 7.9 shows the hue, saturation, and intensity components of an image of an RGB cube on a white background, similar to the image in Fig. 7.2(b). Figure 7.9(a) is the hue image. Its most distinguishing feature is the discontinuity in value along a 45° line in the front (red) plane of the cube. To understand the reason for this discontinuity, refer to Fig. 7.2(b), draw a line from the red to the white vertices of the cube, and select a point in the middle of this line. Starting at that point, draw a path to the right, following the cube around until you return to the starting point. The major colors encountered in this path are yellow, green, cyan, blue, magenta, and back to red. According to Fig. 7.7, the value of hue along this path should increase from 0° to 360° (i.e., from the lowest to highest possible values of hue). This is precisely what Fig. 7.9(a) shows because the lowest value is represented as black and the highest value as white in the figure.



a b c

FIGURE 7.9 HSI component images of an image of an RGB color cube. (a) Hue, (b) saturation, and (c) intensity images.

The saturation image in Fig. 7.9(b) shows progressively darker values toward the white vertex of the RGB cube, indicating that colors become less and less saturated as they approach white. Finally, every pixel in the image shown in Fig. 7.9(c) is the average of the RGB values at the corresponding pixel location in Fig. 7.2(b). Note that the background in this image is white because the intensity of the background in the color image is white. It is black in the other two images because the hue and saturation of white are zero. ■

7.2.6 Device-Independent Color Spaces

The focus of the material in Sections 7.2.1 through 7.2.5 is primarily on color spaces that represent color information in ways that make calculations more convenient, or because they represent colors in ways that are more intuitive or suitable for a particular application. All the spaces discussed thus far are *device-dependent*. For example, the appearance of RGB colors varies with display and scanner characteristics, and CMYK colors vary with printer, ink, and paper characteristics.

The focus of this section is on device-independent color spaces. Achieving consistency and high-quality color reproduction in a color imaging system requires the understanding and characterization of every color device in the system. In a *controlled* environment, it is possible to "tune" the various components of the system to achieve satisfactory results. For example, in a one-shop photographic printing operation, it is possible to optimize manually the color dyes, as well as the development, and printing subsystems to achieve consistent reproduction results. On the other hand, this approach is not practical (or even possible) in *open* digital imaging systems that consist of many devices, or in which there is no control over where images are processed or viewed (e.g., the Internet).

Background

The characteristics used generally to distinguish one color from another are *brightness, hue*, and *saturation*. As indicated earlier in this section, brightness embodies the achromatic notion of intensity. Hue is an attribute associated with the dominant wavelength in a mixture of light waves. Hue represents dominant color as perceived by an observer. Thus, when we call an object red, orange, or yellow, we are referring to its hue. Saturation refers to the relative purity or the amount of white light mixed with a hue. The pure spectrum colors are fully saturated. Colors such as pink (red and white) and lavender (violet and white) are less saturated, with the degree of saturation being inversely proportional to the amount of white light added.

Hue and saturation taken together are called *chromaticity*, and, therefore, a color may be characterized by its brightness and chromaticity. The amounts of red, green, and blue needed to form any particular color are called the *tristimulus values* and are denoted, X, Y, and Z, respectively. A color is then specified by its *trichromatic coefficients*, defined as

$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}$$

and

$$z = \frac{Z}{X + Y + Z} = 1 - x - y$$

It then follows that

$$x + y + z = 1$$

where, x, y, and z represent components of red, green, and blue, respectively.[†] For any wavelength of light in the visible spectrum, the tristimulus values needed to produce the color corresponding to that wavelength can be obtained directly from curves or tables that have been compiled from extensive experimental results (Poynton [1996].

One of the most widely used device-independent tristimulus color spaces is the 1931 CIE XYZ color space, developed by the International Commission on Illumination (known by the acronym CIE, for Commission Internationale de l'Éclairage). In the CIE XYZ color space, Y was selected specifically to be a measure of brightness. The color space defined by Y and the chromaticity values x and y is called the CIE xyY color space. The X and Z tristimulus values can be computed from the x, y, and Y values using the following equations:

$$X = \frac{Y}{y} x$$

and

$$Z = \frac{Y}{y}(1 - x - y)$$

You can see from the preceding equations that there is a direct correspondence between the XYZ and xyY CIE color spaces.

A diagram (Fig. 7.10) showing the range of colors perceived by humans as a function of x and y is called a *chromaticity diagram*. For any values of x and y in the diagram, the corresponding value of z is $z = 1 - (x + y)$. For example the point marked green in Fig. 7.10 has approximately 62% green and 25% red, so the blue component of light for that color is 13%.

[†]The use of x, y, and z to denote chromaticity coefficients follows notational convention. These should not be confused with the use of (x, y) to denote spatial coordinates in other sections of the book.

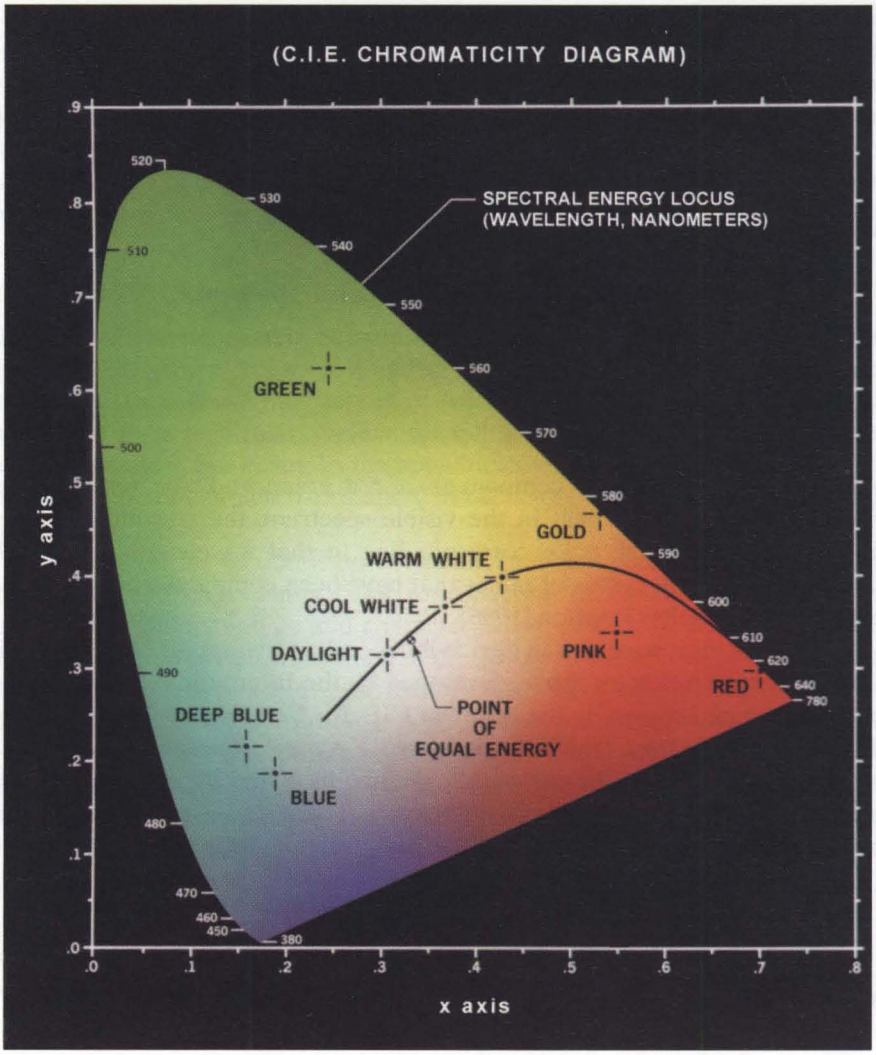


FIGURE 7.10
CIE chromaticity diagram. (Courtesy of the General Electric Co. Lamp Business Division.)

Because of the limitations of display and printing devices, chromaticity diagrams can only approximate the full range of perceptible colors.

The positions of the various monochromatic (pure spectrum) colors—from violet at 380 nm to red at 780 nm—are indicated around the boundary of the tongue-shaped section of the chromaticity diagram. The straight portion of the boundary is called the *line of purples*; these colors do not have a monochromatic equivalent. Any point not actually on the boundary but within the diagram represents some mixture of spectrum colors. The point of equal energy in Fig. 7.10 corresponds to equal fractions of the three primary colors; it represents the CIE standard for white light. Any point located on the boundary of the chromaticity chart is fully saturated. As a point leaves the boundary and approaches the point of equal energy, more white light is added to the color

and it becomes less saturated. The color saturation at the point of equal energy is zero.

A straight-line segment joining any two points in the diagram defines all the different color variations that can be obtained by combining those two colors additively. Consider, for example, a straight line joining the red and green points in Fig. 7.10. If there is more red light than green light in a color, the point representing the color will be on the line segment, closer to the red point than to the green point. Similarly, a line drawn from the point of equal energy to any point on the boundary of the chart will define all the shades of that particular spectrum color.

Extension of this procedure to three colors is straightforward. To determine the range of colors that can be obtained from any three given colors in the chromaticity diagram, we draw connecting lines to each of the three color points. The result is a triangle, and any color on the boundary or inside the triangle can be produced by various combinations of the three initial colors. A triangle with vertices at any three *fixed* colors cannot enclose the *entire* color region in Fig. 7.10. This observation makes it clear that the often-made remark that any color can be generated from three fixed primary colors is a misconception.

The CIE family of device-independent color spaces

In the decades since the introduction of the XYZ color space, the CIE has developed several additional color space specifications that attempt to provide alternative color representations that are better suited to some purposes than XYZ. For example, the CIE introduced in 1976 the L*a*b* color space, which is widely used in color science, creative arts, and the design of color devices such as printers, cameras, and scanners. L*a*b* provides two key advantages over XYZ as a working space. First, L*a*b* more clearly separates gray-scale information (entirely represented as L^* values) from color information (represented using a^* and b^* values). Second, the L*a*b* color was designed so the Euclidean distance in this space corresponds reasonably well with perceived differences between colors. Because of this property, the L*a*b* color space is said to be *perceptually uniform*. As a corollary, L^* values relate linearly to human perception of brightness. That is, if one color has an L^* value twice as large as the L^* value of another, the first color is perceived to be about twice as bright. Note that, because of the complexity of the human visual system, the perceptual uniformity property holds only approximately.

Table 7.4 lists the CIE device-independent color spaces supported by the Image Processing Toolbox. See the book by Sharma [2003] for technical details of the various CIE color models.

The sRGB color space

As mentioned earlier in this section, the RGB color model is device dependent, meaning that there is no single, unambiguous color interpretation for a given set of R, G, and B values. In addition, image files often contain no information about the color characteristics of the device used to capture them. As a result, the same image file could (and often did) look substantially different on different

TABLE 7.4 Device-independent CIE color spaces supported by the Image Processing Toolbox.

Color space	Description
XYZ	The original, 1931 CIE color space specification.
xyY	CIE specification that provides normalized chromaticity values. The capital Y value represents luminance and is the same as in XYZ.
uvL	CIE specification that attempts to make the chromaticity plane more visually uniform. L is luminance and is the same as Y in XYZ.
u′v′L	CIE specification in which u and v are re-scaled to improve uniformity.
L*a*b*	CIE specification that attempts to make the luminance scale more perceptually uniform. L^* is a nonlinear scaling of L, normalized to a reference white point.
L*ch	CIE specification where c is chroma and h is hue. These values are a polar coordinate conversion of a^* and b^* in $L^*a^*b^*$.

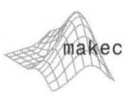
computer systems. As Internet use soared in the 1990s, web designers often found they could not accurately predict how image colors would look when displayed in users' browsers.

To address these issues, Microsoft and Hewlett-Packard proposed a new standard default color space called sRGB (Stokes et al. [1996]). The sRGB color space was designed to be consistent with the characteristics of standard computer CRT monitors, as well as with typical home and office viewing environments for personal computers. The sRGB color space is device independent, so sRGB color values can readily be converted to other device-independent color spaces.

The sRGB standard has become widely accepted in the computer industry, especially for consumer-oriented devices. Digital cameras, scanners, computer displays, and printers are routinely designed to assume that image RGB values are consistent with the sRGB color space, unless the image file contains more specific device color information.

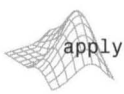
CIE and sRGB color space conversions

The toolbox functions `makecform` and `applycform` can be used to convert between several device-independent color spaces. Table 7.5 lists the conversions supported. Function `makecform` creates a cform structure, similar to the way `maketform` creates a tform structure (see Chapter 6). The relevant `makecform` syntax is:




```
cform = makecform(type)
```

where `type` is one of the strings shown in Table 7.5. Function `applycform` uses the cform structure to convert colors. The `applycform` syntax is:




```
g = applycform(f, cform)
```

types used in makecform	Color spaces
'lab2lch', 'lch2lab'	L*a*b* and L*ch
'lab2srgb', 'srgb2lab'	L*a*b* and sRGB
'lab2xyz', 'xyz2lab'	L*a*b* and XYZ
'srgb2xyz', 'xyz2srgb'	sRGB and XYZ
'upvpl2xyz', 'xyz2upvpl'	u′v′L and XYZ
'uvl2xyz', 'xyz2uvl'	uvL and XYZ
'xyl2xyz', 'xyz2xyl'	xyY and XYZ

TABLE 7.5 Device-independent color-space conversions supported by the Image Processing Toolbox.

EXAMPLE 7.3: Creating a perceptually uniform color scale based on the L*a*b* color space.

■ In this example we construct a color scale that can be used in both color and gray-scale publications. McNames [2006] lists several principles for designing such a color scale.

1. The perceived difference between two scale colors should be proportional to the distance between them along the scale.
2. Luminance should increase monotonically, so that the scale works for gray-scale publications.
3. Neighboring colors throughout the scale should be as distinct as possible.
4. The scale should encompass a wide range of colors.
5. The color scale should be intuitive.

We will design our color scale to satisfy the first four principles by creating a path through L*a*b* space. The first principle, perceptual scale uniformity, can be satisfied using an equidistant spacing of colors in L*a*b*. The second principle, monotonically increasing luminance, can be satisfied by constructing a linear ramp of L^* values [L^* varies between 0 (black) and 100 (the brightness of a perfect diffuser)]. Here we make a ramp of 1024 values space equally between 40 and 80.

```
>> L = linspace(40, 80, 1024);
```

The third principle, distinct neighboring colors, can be satisfied by varying colors in hue, which corresponds to the polar angle of color coordinates in the a^*b^*-plane.

```
>> radius = 70;
>> theta = linspace(0, pi, 1024);
>> a = radius * cos(theta);
>> b = radius * sin(theta);
```

The fourth principle calls for using a wide range of colors. Our set of a^* and b^* values ranges as far apart (in polar angle) as possible, without the last color in the scale starting to get closer to the first color.



FIGURE 7.11 A perceptually uniform color scale based on the L*a*b* color space.

Next we make a $100 \times 1024 \times 3$ image of the L*a*b* color scale.

```
>> L = repmat(L, 100, 1);
>> a = repmat(a, 100, 1);
>> b = repmat(b, 100, 1);
>> lab_scale = cat(3, L, a, b);
```

To display the color scale image in MATLAB, we first must convert to RGB. We start by making the appropriate cform structure using `makecform`, and then we use `applycform`:

```
>> cform = makecform('lab2srgb');
>> rgb_scale = applycform(lab_scale, cform);
>> imshow(rgb_scale)
```

Figure 7.11 shows the result.

The fifth principle, intuitiveness, is much harder to assess and depends on the application. Different color scales can be constructed using a similar procedure but using different starting and ending values in L^*, as well as in the a^*b^*-plane. The resulting new color scales might be more intuitive for certain applications. ■

ICC color profiles

Document colors can have one appearance on a computer monitor and quite a different appearance when printed. Or the colors in a document may appear different when printed on different printers. In order to obtain high-quality color reproduction between different input, output, and display devices, it is necessary to create a transform to map colors from one device to another. In general, a separate color transform would be needed between every pair of devices. Additional transforms would be needed for different printing conditions, device quality settings, etc. Each of the many transforms would have to be developed using carefully-controlled and calibrated experimental conditions. Clearly such an approach would prove impractical for all but the most expensive, high-end systems.

The International Color Consortium (ICC), an industry group founded in 1993, has standardized a different approach. Each device has just two transforms associated with it, regardless of the number of other devices that may be present in the system. One of the transforms converts device colors to a standard, device-independent color space called the *profile connection space* (PCS). The other transform is the inverse of the first; it converts PCS colors

back to device colors. (The PCS can be either XYZ or L*a*b*.) Together, the two transforms make up the *ICC color profile* for the device.

On of the primary goals of the ICC has been to create, standardize, maintain, and promote the ICC color profile standard (ICC [2004]). The Image Processing Toolbox function `iccread` reads profile files. The `iccread` syntax is:




```
p = iccread(filename)
```

The output, `p`, is a structure containing file header information and the numerical coefficients and tables necessary to compute the color space conversions between device and PCS colors.

Converting colors using ICC profiles is done using `makecform` and `applycform`. The ICC profile syntax for `makecform` is:

```
cform = makecform('icc', src_profile, dest_profile)
```

where `src_profile` is the file name of the source device profile, and `dest_profile` is the file name of the destination device profile.

The ICC color profile standard includes mechanisms for handling a critical color conversion step called *gamut mapping*. A *color gamut* is a volume in color space defining the range of colors that a device can reproduce (CIE [2004]). Color gamuts differ from device to device. For example, the typical monitor can display some colors that cannot be reproduced using a printer. Therefore it is necessary to take differing gamuts into account when mapping colors from one device to another. The process of compensating for differences between source and destination gamuts is called *gamut mapping* (ISO [2004]).

There are many different methods used for gamut mapping (Morovic [2008]). Some methods are better suited for certain purposes than others. The ICC color profile standard defines four "purposes" (called *rendering intents*) for gamut mapping. These rendering intents are described in Table 7.6. The `makecform` syntax for specifying rendering intents is:

```
cform = makecform('icc', src_profile, dest_profile, ...
        'SourceRenderingIntent', src_intent, ...
          'DestRenderingIntent', dest_intent)
```

where `src_intent` and `dest_intent` are chosen from the strings `'Perceptual'` (the default), `'AbsoluteColorimetric'`, `'RelativeColorimetric'`, and `'Saturation'`.

EXAMPLE 7.4: Soft proofing using ICC color profiles.

■ In this example we use ICC color profiles, `makecform`, and `applycform` to implement a process called *soft proofing*. Soft proofing simulates on a computer monitor the appearance that a color image would have if printed. Conceptually, soft proofing is a two-step process:

1. Convert monitor colors (often assuming sRGB) to output device colors, usually using the perceptual rendering intent.

TABLE 7.6
ICC profile rendering intents.

Rendering intent	Description
Perceptual	Optimizes gamut mapping to achieve the most aesthetically pleasing result. In-gamut colors might not be maintained.
Absolute colorimetric	Maps out-of-gamut colors to the nearest gamut surface. Maintains relationship of in-gamut colors. Renders colors with respect to a perfect diffuser.
Relative colorimetric	Maps out-of-gamut colors to the nearest gamut surface. Maintains relationship of in-gamut colors. Renders colors with respect to the white point of the device or output media.
Saturation	Maximizes saturation of device colors, possibly at the expense of shifting hue. Intended for simple graphic charts and graphs, rather than images.

2. Convert the computed output device colors back to monitor colors, using the absolute colorimetric rendering intent.

For our input profile we will use `sRGB.icm`, a profile representing the sRGB color space that ships with the toolbox. Our output profile is `SNAP2007.icc`, a newsprint profile contained in the ICC's profile registry (www.color.org/registry). Our sample image is the same as in Fig. 7.4(a).

We first preprocess the image by adding a thick white border and a thin gray border around the image. These borders will make it easier to visualize the simulated "white" of the newsprint.

```
>> f = imread('Fig0704(a).tif');
>> fp = padarray(f, [40 40], 255, 'both');
>> fp = padarray(fp, [4 4], 230, 'both');
>> imshow(fp)
```

Figure 7.12(a) shows the padded image.

Next we read in the two profiles and use them to convert the iris image from sRGB to newsprint colors.

```
>> p_srgb = iccread('sRGB.icm');
>> p_snap = iccread('SNAP2007.icc');
>> cform1 = makecform('icc', p_srgb, p_snap);
>> fp_newsprint = applycform(fp, cform1);
```

Finally we create a second `cform` structure, using the absolute colorimetric rendering intent, to convert back to sRGB for display.

```
>> cform2 = makecform('icc', p_snap, p_srgb, ...
        'SourceRenderingIntent', 'AbsoluteColorimetric', ...
        'DestRenderingIntent', 'AbsoluteColorimetric');
```



a b

FIGURE 7.12
Soft proofing example. (a) Original image with white border. (b) Simulation of image appearance when printed on newsprint.

```
>> fp_proof = applycform(fp_newsprint, cform2);
>> imshow(fp_proof)
```

Figure 7.12(b) shows the result. This figure itself is only an approximation of the result as actually seen on a monitor because the color gamut of this printed book is not the same as the monitor gamut. ■

7.3 The Basics of Color Image Processing

In this section we begin the study of processing techniques applicable to color images. Although they are far from being exhaustive, the techniques developed in the sections that follow are illustrative of how color images are handled for a variety of image-processing tasks. For the purposes of the following discussion we subdivide color image processing into three principal areas: (1) *color transformations* (also called *color mappings*); (2) *spatial processing* of individual color planes; and (3) *color vector processing*. The first category deals with processing the pixels of each color plane based strictly on their values and not on their spatial coordinates. This category is analogous to the material in Section 3.2 dealing with intensity transformations. The second category deals with spatial (neighborhood) filtering of *individual* color planes and is analogous to the discussion in Sections 3.4 and 3.5 on spatial filtering.

The third category deals with techniques based on processing all components of a color image simultaneously. Because full-color images have at least three components, color pixels can be treated as vectors. For example, in the RGB system, each color point can be interpreted as a vector extending from the origin to that point in the RGB coordinate system (see Fig. 7.2).

Let $\mathbf{c}$ represent an arbitrary vector in RGB color space:

$$\mathbf{c} = \begin{bmatrix} c_R \\ c_G \\ c_B \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

This equation indicates that the components of $\mathbf{c}$ are simply the RGB components of a color image *at a point*. We take into account the fact that the color components are a function of coordinates by using the notation

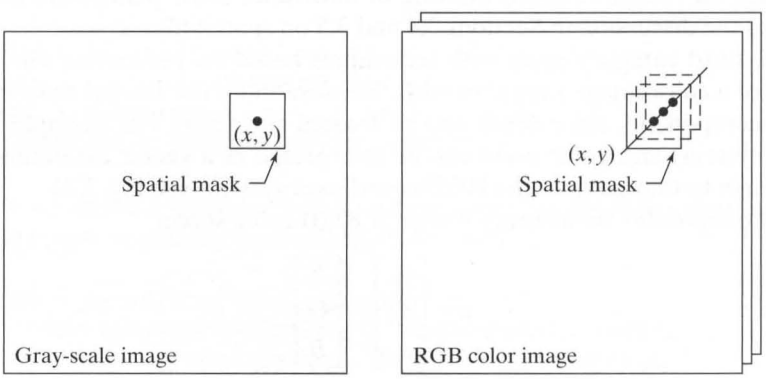
$$\mathbf{c}(x,y) = \begin{bmatrix} c_R(x,y) \\ c_G(x,y) \\ c_B(x,y) \end{bmatrix} = \begin{bmatrix} R(x,y) \\ G(x,y) \\ B(x,y) \end{bmatrix}$$

For an image of size $M \times N$ there are MN such vectors, $\mathbf{c}(x,y)$, for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$.

In some cases, equivalent results are obtained whether color images are processed one plane at a time or as vector quantities. However, as explained in more detail in Section 7.6, this is not always the case. In order for the two approaches to be equivalent, two conditions have to be satisfied: First, the process has to be applicable to both vectors and scalars. Second, the operation on each component of a vector must be independent of the other components. As an illustration, Fig. 7.13 shows spatial neighborhood processing of gray-scale and full-color images. Suppose that the process is neighborhood averaging. In Fig. 7.13(a), averaging would be accomplished by summing the gray levels of all the pixels in the neighborhood and dividing by the total number of pixels in the neighborhood. In Fig. 7.13(b) averaging would be done by summing all the vectors in the neighborhood and dividing each component by the total number of vectors in the neighborhood. But each component of the average vector is the sum of the pixels in the image corresponding to that component, which is the same as the result that would be obtained if the averaging were done on the neighborhood of each color component image individually, and then the color vector were formed.

7.4 Color Transformations

The techniques described in this section are based on processing the color components of a color image or intensity component of a monochrome image within the context of a single color model. For color images, we restrict attention to transformations of the form



a b

FIGURE 7.13 Spatial masks for (a) gray-scale and (b) RGB color images.

$$s_i = T_i(r_i) \qquad i = 1, 2, \ldots, n$$

where r_i and s_i are the color components of the input and output images, n is the dimension of (or number of color components in) the color space of r_i and the T_i are referred to as *full-color transformation* (or *mapping*) *functions*.

If the input images are monochrome, then we write an equation of the form

$$s_i = T_i(r) \qquad i = 1, 2, \ldots, n$$

where r denotes gray-level values, s_i and T_i are as above, and n is the number of color components in s_i. This equation describes the mapping of gray levels into arbitrary colors, a process frequently referred to as a *pseudocolor transformation* or *pseudocolor mapping*. Note that the first equation can be used to process monochrome images if we let $r_1 = r_2 = r_3 = r$. In either case, the equations given here are straightforward extensions of the intensity transformation equation introduced in Section 3.2. As is true of the transformations in that section, all n pseudo- or full-color transformation functions $\{T_1, T_2, \ldots, T_n\}$ are independent of the spatial image coordinates (x, y).

Some of the gray-scale transformations introduced in Chapter 3, such as `imcomplement`, which computes the negative of an image, are independent of the gray-level content of the image being transformed. Others, like `histeq`, which depends on gray-level distribution, are adaptive, but the transformation is fixed once its parameters have been estimated. And still others, like `imadjust`, which requires the user to select appropriate curve shape parameters, are often best specified interactively. A similar situation exists when working with pseudo- and full-color mappings—particularly when human viewing and interpretation (e.g., for color balancing) are involved. In such applications, the selection of appropriate mapping functions is best accomplished by directly manipulating graphical representations of candidate functions and viewing their combined effect (in real time) on the images being processed.

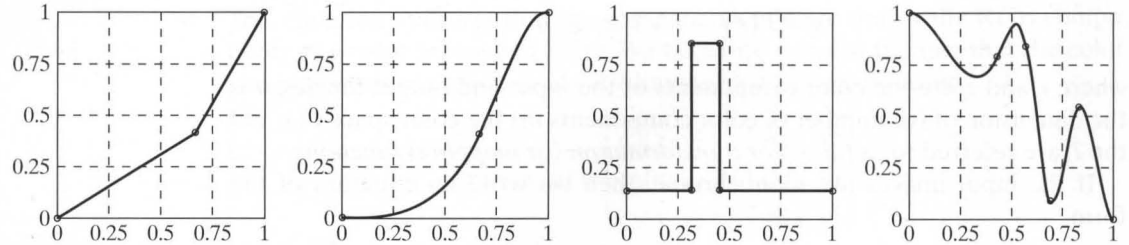
Figure 7.14 illustrates a simple but powerful way to specify mapping functions graphically. Figure 7.14(a) shows a transformation that is formed by linearly interpolating three *control points* (the circled coordinates in the figure); Fig. 7.14(b) shows the transformation that results from a cubic spline interpolation of the same three points; and Figs. 7.14(c) and (d) provide more complex linear and cubic spline interpolations, respectively. Both types of interpolation are supported in MATLAB. Linear interpolation is implemented by using

```
z = interp1q(x, y, xi)
```



`interp1q`

which returns a column vector containing the values of the linearly interpolated 1-D function `z` at points `xi`. Column vectors `x` and `y` specify the coordinates of the underlying control points. The elements of `x` must increase monotonically. The length of `z` is equal to the length of `xi`. Thus, for example,



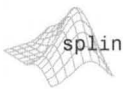
a b c d

FIGURE 7.14 Specifying mapping functions using control points: (a) and (c) linear interpolation and (b) and (d) cubic spline interpolation.

```
>> z = interp1q([0 255]', [0 255]', [0: 255]')
```

produces a 256-element one-to-one mapping connecting control points (0, 0) and (255, 255)—that is, `z = [0 1 2 . . . 255]'`.

In a similar manner, cubic spline interpolation is implemented using the spline function,



```
z = spline(x, y, xi)
```

where variables `z`, `x`, `y`, and `xi` are as described in the previous paragraph for `interp1q`. However, the `xi` must be distinct for use in function `spline`. Moreover, if `y` contains two more elements than `x`, its first and last entries are assumed to be the end slopes of the cubic spline. The function depicted in Fig. 7.14(b), for example, was generated using zero-valued end slopes.

The specification of transformation functions can be made interactive by graphically manipulating the control points that are input to functions `interp1q` and `spline` and displaying in real time the results of the images being processed. Custom function `ice` (interactive color editing) does precisely this. Its syntax is

`ice`

The development of function `ice`, given in Appendix B, is a comprehensive illustration of how to design a graphical user interface (GUI) in MATLAB.

```
g = ice('Property Name', 'Property Value', . . .)
```

where `'Property Name'` and `'Property Value'` must appear in pairs, and the dots indicate repetitions of the pattern consisting of corresponding input pairs. Table 7.7 lists the valid pairs for use in function `ice`. Some examples are given later in this section.

With reference to the `'wait'` parameter, when the `'on'` option is selected either explicitly or by default, the output `g` is the processed image. In this case, `ice` takes control of the process, including the cursor, so nothing can be typed on the command window until the function is closed, at which time the final result is an image with handle `g` (or any graphics object in general). When `'off'` is selected, `g` is the *handle*[†] of the processed image, and control is returned

[†]Whenever MATLAB creates a graphics object, it assigns an identifier (called a *handle*) to the object, used to access the object's properties. Graphics handles are useful when modifying the appearance of graphs or creating custom plotting commands by writing M-files that create and manipulate objects directly. The concept is discussed in Sections 2.10.4, 2.10.5, and 3.3.1.

Property Name	Property Value
'image'	An RGB or monochrome input image, f, to be transformed by interactively-specified mappings.
'space'	The color space of the components to be modified. Possible values are 'rgb', 'cmy', 'hsi', 'hsv', 'ntsc' (or 'yiq'), and 'ycbcr'. The default is 'rgb'.
'wait'	If 'on' (the default), g is the mapped input image. If 'off', g is the handle of the mapped input image.

TABLE 7.7
Valid inputs for function ice.

immediately to the command window; therefore, new commands can be typed with the ice function still active. To obtain the properties of a graphics object we use the get function

```
h = get(g)
```

See the discussion of formats in Section 2.10.2 for another syntax of function get.

This function returns all properties and applicable current values of the graphics object identified by the handle g. The properties are stored in structure h, so typing h at the prompt lists all the properties of the processed image (see Section 2.10.7 for an explanation of structures). To extract a particular property, we type h.PropertyName.

Letting f denote an RGB or monochrome image, the following are examples of the syntax of function ice:

```
>> ice                                    % Only the ice
                                          % graphical
                                          % interface is
                                          % displayed.
>> g = ice('image', f);                   % Shows and returns
                                          % the mapped image g.
>> g = ice('image', f, 'wait', 'off')     % Shows g and returns
                                          % the handle.
>> g = ice('image', f, 'space', 'hsi')    % Maps RGB image f in
                                          % HSI space.
```

Note that when a color space other than RGB is specified, the input image (whether monochrome or RGB) is transformed to the specified space before any mapping is performed. The mapped image is then converted to RGB for output. The output of ice is always RGB; its input is always monochrome or RGB. If we type g = ice('image', f), an image and graphical user interface (GUI) like that shown in Fig. 7.15 appear on the MATLAB desktop. Initially, the transformation curve is a straight line with a control point at each end. Control points are manipulated with the mouse, as summarized in Table 7.8. Table 7.9 lists the function of the other GUI components. The following examples show typical applications of function ice.

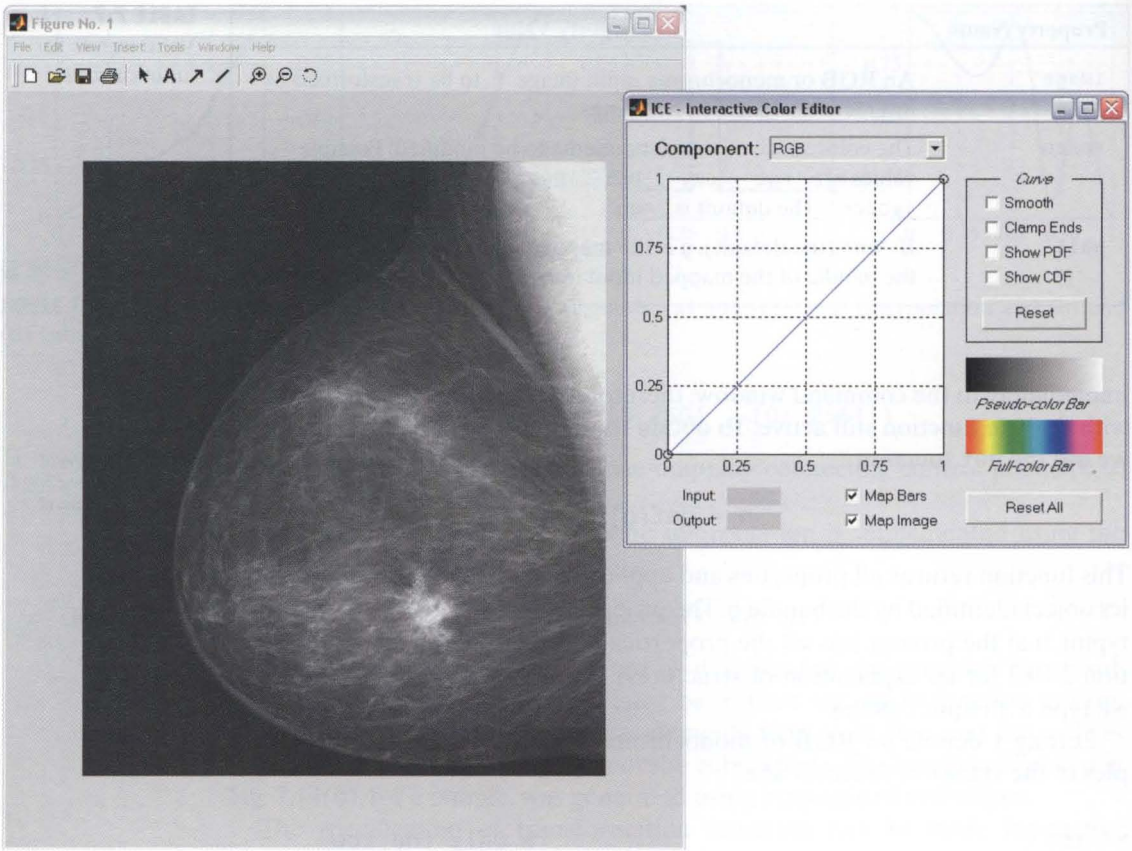


FIGURE 7.15 The typical opening windows of function `ice`. (Image courtesy of G.E. Medical Systems.)

EXAMPLE 7.5: Inverse mappings: monochrome negatives and color complements.

■ Figure 7.16(a) shows the `ice` interface with the default RGB curve of Fig. 7.15 modified to produce an inverse or negative mapping function. To create the new mapping function, control point $(0, 0)$ is moved (by clicking and dragging it to the upper-left corner) to $(0, 1)$ and control point $(1, 1)$ is moved to coordinate $(1, 0)$. Note how the coordinates of the cursor are displayed in red in the Input/Output boxes. Only the RGB map is modified; the individual R, G, and B

TABLE 7.8 Manipulating control points with the mouse.

Mouse action[†]	Result
Left Button	Move control point by pressing and dragging..
Left Button + Shift Key	Add control point. The location of the control point can be changed by dragging (while still pressing the Shift Key).
Left Button + Control Key	Delete control point.

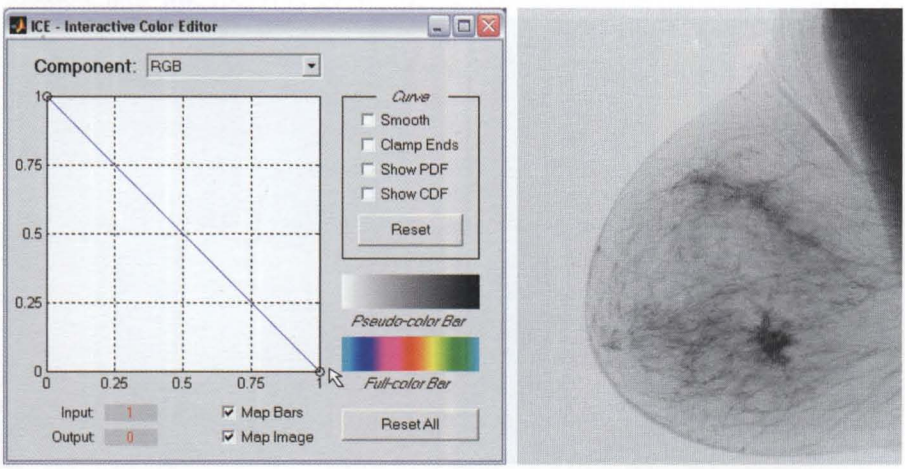
[†] For three button mice, the left, middle, and right buttons correspond to the move, add, and delete operations in the table.

TABLE 7.9 Function of the checkboxes and pushbuttons in the `ice` GUI.

GUI Element	Description
Smooth	Checked for cubic spline (smooth curve) interpolation. If unchecked, piecewise linear interpolation is used.
Clamp Ends	Checked to force the starting and ending curve slopes in cubic spline interpolation to 0. Piecewise linear interpolation is not affected.
Show PDF	Display probability density function(s) [i.e., histogram(s)] of the image components affected by the mapping function.
Show CDF	Display cumulative distribution function(s) instead of PDFs. (Note: PDFs and CDFs cannot be displayed simultaneously.)
Map Image	If checked, image mapping is enabled; otherwise it is not.
Map Bars	If checked, pseudo- and full-color bar mapping is enabled; otherwise the unmapped bars (a gray wedge and hue wedge, respectively) are displayed.
Reset	Initialize the currently displayed mapping function and uncheck all curve parameters.
Reset All	Initialize all mapping functions.
Input/Output	Show the coordinates of a selected control point on the transformation curve. Input refers to the horizontal axis, and Output to the vertical axis.
Component	Select a mapping function for interactive manipulation. In RGB space, possible selections include R, G, B, and RGB (which maps all three color components). In HSI space, the options are H, S, I, and HSI, and so on.

maps are left in their 1:1 default states (see the Component entry in Table 7.6). For monochrome inputs, this guarantees monochrome outputs. Figure 7.16(b) shows the monochrome negative that results from the inverse mapping. Note that it is identical to Fig. 3.3(b), which was obtained using the `imcomplement` function. The pseudocolor bar in Fig. 7.16(a) is the "photographic negative" of the original gray-scale bar in Fig. 7.15.

Default (i.e., 1:1) mappings are not shown in most examples.



a b

FIGURE 7.16
(a) A negative mapping function, and (b) its effect on the monochrome image of Fig. 7.15.

Inverse or negative mapping functions also are useful in color processing. As shown in Figs. 7.17(a) and (b), the result of the mapping is reminiscent of conventional color film negatives. For instance, the red stick of chalk in the bottom row of Fig. 7.17(a) is transformed to cyan in Fig. 7.17(b)—the *color complement* of red. The complement of a primary color is the mixture of the other two primaries (e.g., cyan is blue plus green). As in the gray-scale case, color complements are useful for enhancing detail that is embedded in dark regions of color—particularly when the regions are dominant in size. Note that the *Full-color Bar* in Fig. 7.16(a) contains the complements of the hues in the *Full-color Bar* of Fig. 7.15. ■

EXAMPLE 7.6: Monochrome and color contrast enhancement.

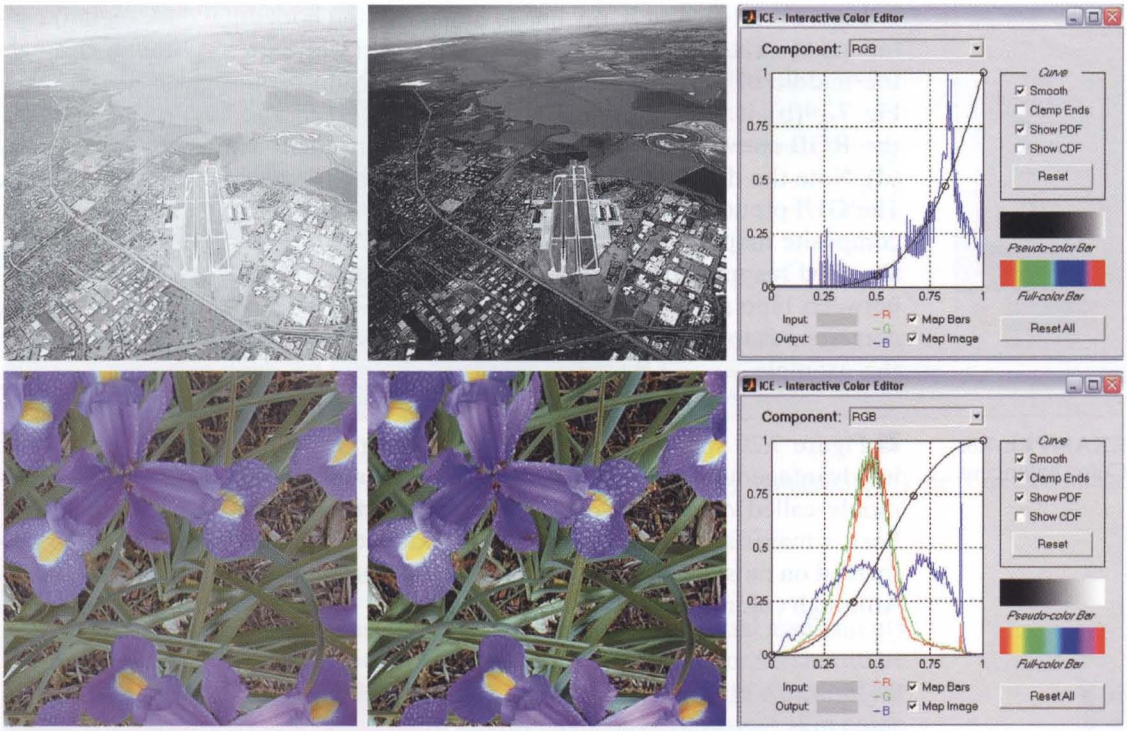
■ Consider next the use of function `ice` for monochrome and color contrast manipulation. Figures 7.18(a) through (c) demonstrate the effectiveness of `ice` in processing monochrome images. Figures 7.18(d) through (f) show similar effectiveness for color inputs. As in the previous example, mapping functions that are not shown remain in their default or 1:1 state. In both processing sequences, the Show PDF check box is enabled. Thus, the histogram of the aerial photo in (a) is displayed under the gamma-shaped mapping function (see Section 3.2.1) in (c); and three histograms are provided in (f) for the color image in (c)—one for each of its three color components. Although the S-shaped mapping function in (f) increases the contrast of the image in (d) [compare it to (e)], it also has a slight effect on hue. The small change of color is virtually imperceptible in (e), but is an obvious result of the mapping, as can be seen in the mapped full-color reference bar in (f). Recall from the previous example that equal changes to the three components of an RGB image can have a dramatic effect on color (see the color complement mapping in Fig. 7.17). ■

The red, green, and blue components of the input images in Examples 7.5 and 7.6 are mapped identically—that is, using the same transformation function. To



a b

FIGURE 7.17 (a) A full color image, and (b) its negative (color complement).



a b c
d e f

FIGURE 7.18 Using function `ice` for monochrome and full color contrast enhancement: (a) and (d) are the input images, both of which have a "washed-out" appearance; (b) and (e) show the processed results; (c) and (f) are the `ice` displays. (Original monochrome image for this example courtesy of NASA.)

avoid the specification of three identical functions, function `ice` provides an "all components" function (the RGB curve when operating in the RGB color space) that is used to map all input components. The remaining examples in this section demonstrate transformations in which the three components are processed differently.

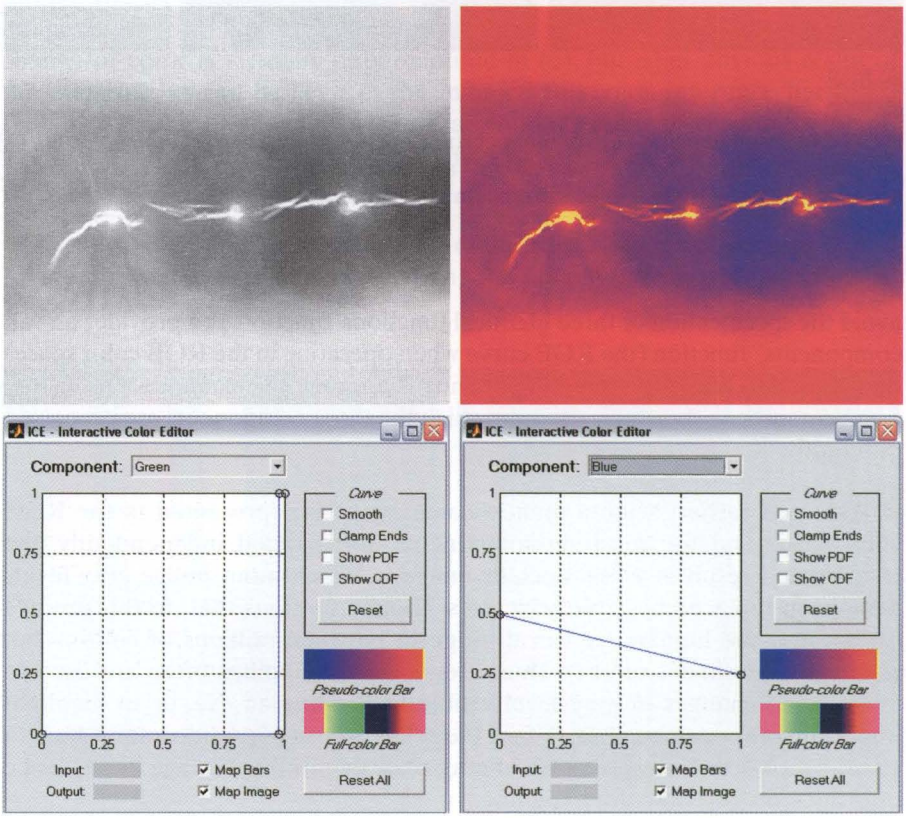
EXAMPLE 7.7: Pseudocolor mappings.

■ As noted earlier, when a monochrome image is represented in the RGB color space and the resulting components are mapped independently, the transformed result is a pseudocolor image in which input image gray levels have been replaced by arbitrary colors. Transformations that do this are useful because the human eye can distinguish between millions of colors—but relatively few shades of gray. Thus, pseudocolor mappings often are used to make small changes in gray level visible to the human eye, or to highlight important gray-scale regions. In fact, the principal use of pseudocolor is human visualization—the interpretation of gray-scale events in an image or sequence of images via gray-to-color assignments.

Figure 7.19(a) is an X-ray image of a weld (the horizontal dark region) containing several cracks and porosities (the bright white streaks running through the middle of the image). A pseudocolor version of the image in shown in Fig. 7.19(b); it was generated by mapping the green and blue components of the RGB-converted input using the mapping functions in Figs. 7.19(c) and (d). Note the dramatic visual difference that the pseudocolor mapping makes. The GUI pseudocolor reference bar provides a convenient visual guide to the composite mapping. As you can see in Figs. 7.19(c) and (d), the interactively specified mapping functions transform the black-to-white gray scale to hues between blue and red, with yellow reserved for white. The yellow, of course, corresponds to weld cracks and porosities, which are the important features in this example. ■

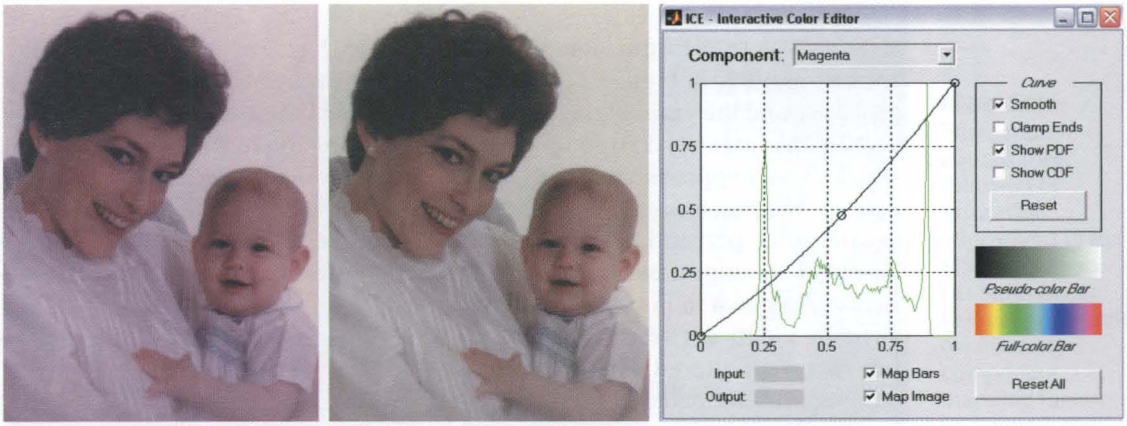
EXAMPLE 7.8: Color balancing.

■ Figure 7.20 shows an application involving a full-color image, in which it is advantageous to map an image's color components independently. Commonly called *color balancing* or *color correction*, this type of mapping has been a mainstay of high-end color reproduction systems but now can be performed on most desktop computers. One important use is photo enhancement. Although color imbalances can be determined objectively by analyzing—with



a b
c d

FIGURE 7.19
(a) X-ray of a defective weld; (b) a pseudocolor version of the weld; (c) and (d) mapping functions for the green and blue components. (Original image courtesy of X-TEK Systems, Ltd.)



a b c

FIGURE 7.20 Using function `ice` for color balancing: (a) an image heavy in magenta; (b) the corrected image; and (c) the mapping function used to correct the imbalance.

a color spectrometer—a known color in an image, accurate visual assessments are possible when white areas, where the RGB or CMY components should be equal, are present. As can be seen in Fig. 7.20, skin tones also are excellent for visual assessments because humans are highly perceptive of proper skin color.

Figure 7.20(a) shows a CMY scan of a mother and her child with an excess of magenta (keep in mind that only an RGB version of the image can be displayed by MATLAB). For simplicity and compatibility with MATLAB, function `ice` accepts only RGB (and monochrome) inputs as well—but can process the input in a variety of color spaces, as detailed in Table 7.7. To interactively modify the CMY components of RGB image `f1`, for example, the appropriate `ice` call is

```
>> f2 = ice('image', f1, 'space', 'CMY');
```

As Fig. 7.20 shows, a small decrease in magenta had a significant impact on image color. ■

EXAMPLE 7.9: Histogram-based mappings.

■ Histogram equalization is a gray-level mapping process that seeks to produce monochrome images with uniform intensity histograms. As discussed in Section 3.3.2, the required mapping function is the cumulative distribution function (CDF) of the gray levels in the input image. Because color images have multiple components, the gray-scale technique must be modified to handle more than one component and associated histogram. As might be expected, it is unwise to histogram equalize the components of a color image independently. The result usually is erroneous color. A more logical approach is to spread color intensities uniformly, leaving the colors themselves (i.e., the hues) unchanged.

Figure 7.21(a) shows a color image of a caster stand containing cruets and shakers. The transformed image in Fig. 7.21(b), which was produced using the transformations in Figs. 7.21(c) and (d), is significantly brighter. Several of the moldings and the grain of the wood table on which the caster is resting are now visible. The intensity component was mapped using the function in Fig. 7.21(c), which closely approximates the CDF of that component (also displayed in the figure). The hue mapping function in Fig. 7.21(d) was selected to improve the overall color perception of the intensity-equalized result. Note that the histograms of the input and output image's hue, saturation, and intensity components are shown in Figs. 7.21(e) and (f), respectively. The hue components are virtually identical (which is desirable), while the intensity and saturation components were altered. Finally note that, to process an RGB image in the HSI color space, we included the input property name/value pair `'space'/'hsi'` in the call to `ice`. ■

The output images generated in the preceding examples in this section are of type RGB and class `uint8`. For monochrome results, as in Example 7.5, all three components of the RGB output are identical. A more compact representation can be obtained via the `rgb2gray` function of Table 7.3 or by using the command

```
>> f3 = f2(:, :, 1);
```

where `f2` is an RGB image generated by `ice`, and `f3` is a monochrome image.

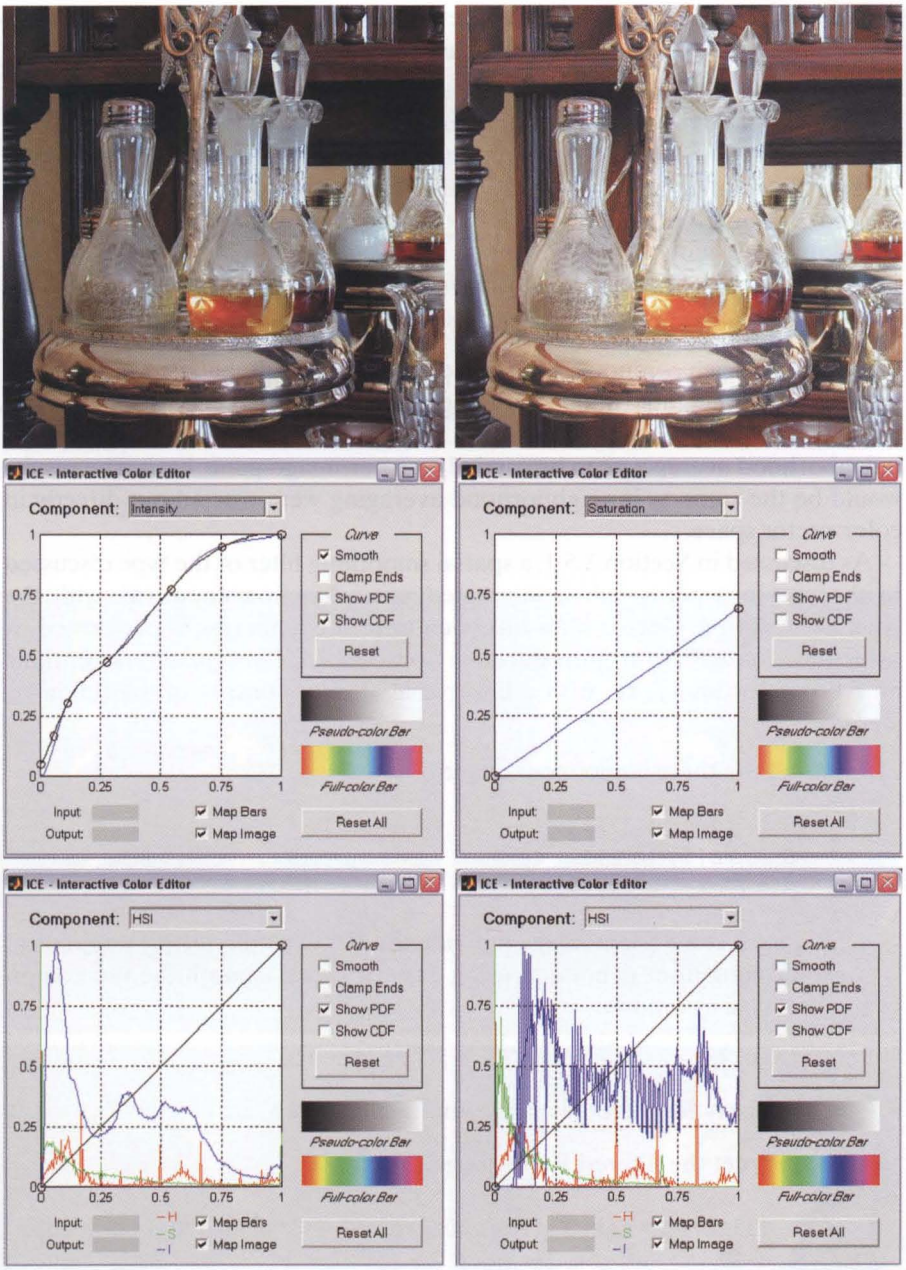
7.5 Spatial Filtering of Color Images

The material in Section 7.4 deals with color transformations performed on single image pixels of single color component planes. The next level of complexity involves performing spatial neighborhood processing, also on single image planes. This breakdown is analogous to the discussion on intensity transformations in Section 3.2, and the discussion on spatial filtering in Sections 3.4 and 3.5. We introduce spatial filtering of color images by concentrating mostly on RGB images, but the basic concepts are applicable (with proper interpretation) to other color models as well. We illustrate spatial processing of color images by two examples of linear filtering: image smoothing and image sharpening.

7.5.1 Color Image Smoothing

With reference to Fig. 7.13(a) and the discussion in Sections 3.4 and 3.5, one way to smooth a monochrome image it to define a filter mask of 1s, multiply all pixel values by the coefficients in the spatial mask, and divide the result by the sum of the elements in the mask. The process of smoothing a full-color image using spatial masks is shown in Fig. 7.13(b).

The process (in RGB space for example) is formulated in the same way as for gray-scale images, except that instead of single pixels we now deal with vector values in the form shown in Section 7.3. Let S_{xy} denote the set of coordinates



a b
c d
e f

FIGURE 7.21 Histogram equalization followed by saturation adjustment in the HSI color space: (a) input image; (b) mapped result; (c) intensity component mapping function and cumulative distribution function; (d) saturation component mapping function; (e) input image's component histograms; and (f) mapped result's component histograms.

defining a neighborhood centered at (x, y) in the color image. The average of the RGB vectors in this neighborhood is

$$\bar{\mathbf{c}}(x, y) = \frac{1}{K} \sum_{(s,t) \in S_{xy}} \mathbf{c}(s, t)$$

where K is the number of pixels in the neighborhood. It follows from the discussion in Section 7.3 and the properties of vector addition that

$$\bar{\mathbf{c}}(x,y) = \begin{bmatrix} \frac{1}{K}\sum_{(s,t)\in S_{xy}} R(s,t) \\ \frac{1}{K}\sum_{(s,t)\in S_{xy}} G(s,t) \\ \frac{1}{K}\sum_{(s,t)\in S_{xy}} B(s,t) \end{bmatrix}$$

We recognize each component of this vector as the result that we would obtain by performing neighborhood averaging on each individual component image, using the filter mask mentioned above.† Thus, we conclude that smoothing by neighborhood averaging can be carried on a per-image-pane basis. The results would be the same as if neighborhood averaging were carried out directly in color vector space.

As discussed in Section 3.5.1, a spatial smoothing filter of the type discussed in the previous paragraph is generated using function `fspecial` with the `'average'` option. Once a filter has been generated, filtering is performed by using function `imfilter`, introduced in Section 3.4.1. *Conceptually*, smoothing an RGB color image, `fc`, with a linear spatial filter consists of the following steps:

1. Extract the three component images:

```
>> fR = fc(:, :, 1);
>> fG = fc(:, :, 2);
>> fB = fc(:, :, 3);
```

2. Filter each component image individually. For example, letting `w` represent a smoothing filter generated using `fspecial`, we smooth the red component image as follows:

```
>> fR_filtered = imfilter(fR, w, 'replicate');
```

   and similarly for the other two component images.

3. Reconstruct the filtered RGB image:

```
>> fc_filtered = cat(3, fR_filtered, fG_filtered, fB_filtered);
```

However, because we can perform linear filtering of RGB images directly in MATLAB using the same syntax employed for monochrome images, the preceding three steps can be combined into one:

†We used an averaging mask of 1s to simplify the explanation. For an averaging mask whose coefficients are not all equal (e.g., a Gaussian mask) we arrive at the same conclusion by multiplying the color vectors by the coefficients of the mask, adding the results, and letting K be equal to the sum of the mask coefficients.

```
>> fc_filtered = imfilter(fc, w, 'replicate');
```

EXAMPLE 7.10: Color image smoothing.

■ Figure 7.22(a) shows an RGB image of size 1197×1197 pixels and Figs. 7.22(b) through (d) are its RGB component images, extracted using the procedure described in the previous paragraph. We know from the results in the preceding discussion that smoothing the individual component images and forming a composite color image will be same as smoothing the original RGB image using the command given at the end of previous paragraph. Figure 7.24(a) shows the result obtained using an averaging filter of size 25×25 pixels.

Next, we investigate the effects of smoothing only the intensity component of the HSI version of Fig. 7.22(a). Figures 7.23(a) through (c) show the three HSI component images obtained using function `rgb2hsi`, where `fc` is Fig. 7.22(a)

```
>> h = rgb2hsi(fc);
```



a b
c d

FIGURE 7.22 (a) RGB image. (b) through (d) The red, green and blue component images, respectively.



a b c

FIGURE 7.23 From left to right: hue, saturation, and intensity components of Fig. 7.22(a).

```
>> H = h(:, :, 1);
>> S = h(:, :, 2);
>> I = h(:, :, 3);
```

Next, we filter the intensity component using the same filter of size 25×25 pixels. The averaging filter was large enough to produce significant blurring. A filter of this size was selected to demonstrate the difference between smoothing in RGB space and attempting to achieve a similar result using only the intensity component of the image after it had been converted to HSI. Figure 7.24(b) was obtained using the commands:

```
>> w = fspecial('average', 25);
>> I_filtered = imfilter(I, w, 'replicate');
```



a b c

FIGURE 7.24 (a) Smoothed RGB image obtained by smoothing the R, G, and B image planes separately. (b) Result of smoothing only the intensity component of the HSI equivalent image. (c) Result of smoothing all three HSI components equally.

```
>> h = cat(3, H, S, I_filtered);
>> f = hsi2rgb(h); % Back to RGB for comparison.
>> imshow(f);
```

Clearly, the two filtered results are quite different. For example, in addition to the image being less blurred, note the faint green border on the top part of the flower in Fig. 7.24(b). The reason for this is that the hue and saturation components were not changed while the variability of values of the intensity components was reduced significantly by the smoothing process. A logical thing to try would be to smooth all three HSI components using the same filter. However, this would change the relative relationship between values of the hue and saturation and would produce even worse results, as Fig. 7.24(c) shows. Observe in particular how much brighter the green border around the flowers is in this image. This effect is quite visible also around the borders of the center yellow region.

Because all the components of the HSI image were filtered simultaneously, Fig. 7.24(c) was generated using a single call to imfilter: `hFilt = imfilter(h, w, 'replicate');` Image `hFilt` was then converted to RGB and displayed

In general, as the size of the mask decreases, the differences obtained when filtering the RGB component images and the intensity component of the HSI equivalent image also decrease. ■

7.5.2 Color Image Sharpening

Sharpening an RGB color image with a linear spatial filter follows the same procedure outlined in the previous section, but using a sharpening filter instead. In this section we consider image sharpening using the Laplacian (see Section 3.5.1). From vector analysis, we know that the Laplacian of a vector is defined as a vector whose components are equal to the Laplacian of the individual scalar components of the input vector. In the RGB color system, the Laplacian of vector $\mathbf{c}$ introduced in Section 7.3 is

$$\nabla^2[c(x,y)] = \begin{bmatrix} \nabla^2 R(x,y) \\ \nabla^2 G(x,y) \\ \nabla^2 B(x,y) \end{bmatrix}$$

which, as in the previous section, tells us that we can compute the Laplacian of a full-color image by computing the Laplacian of each component image separately.

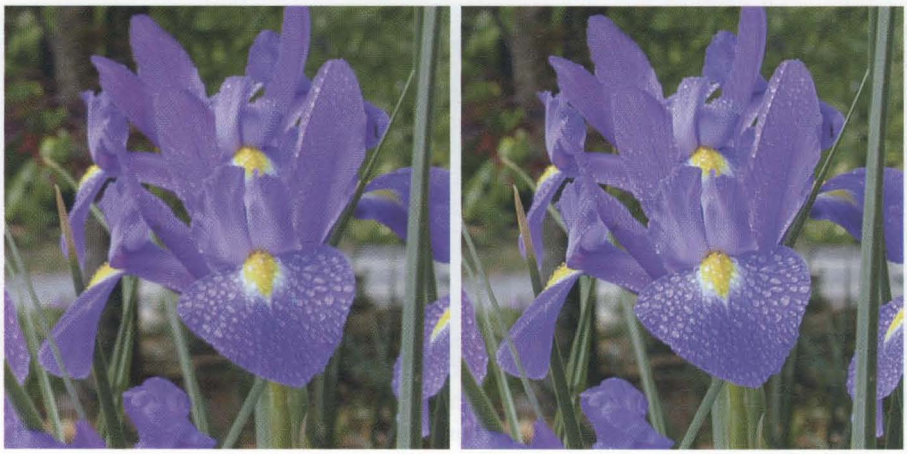
EXAMPLE 7.11 Color image sharpening.

■ Figure 7.25(a) shows a slightly blurred version, `fb`, of the image in Fig. 7.22(a), obtained using a 5×5 averaging filter. To sharpen this image we used the Laplacian (see Section 3.5.1) filter mask

```
>> lapmask = [1 1 1; 1 -8 1; 1 1 1];
```

Then, the enhanced image was computed and displayed using the commands

```
>> fb = tofloat(fb);
```



a b

FIGURE 7.25
(a) Blurred image. (b) Image enhanced using the Laplacian.

```
>> fen = fb - imfilter(fb, lapmask, 'replicate');
>> imshow(fen)
```

As in the previous section, note that the RGB image was filtered directly using `imfilter`. Figure 7.25(b) shows the result. Note the significant increase in sharpness of features such as the water droplets, the veins in the leaves, the yellow centers of the flowers, and the green vegetation in the foreground. ■

7.6 Working Directly in RGB Vector Space

As mentioned in Section 7.3, there are cases in which processes based on individual color planes are not equivalent to working directly in RGB vector space. This is demonstrated in this section, where we illustrate vector processing by considering two important applications in color image processing: color edge detection and region segmentation.

7.6.1 Color Edge Detection Using the Gradient

The gradient of a 2-D function $f(x, y)$ is defined as the vector

$$\nabla \mathbf{f} = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}$$

The magnitude of this vector is

$$\begin{aligned} \nabla f &= \text{mag}(\nabla \mathbf{f}) = \left[g_x^2 + g_y^2 \right]^{1/2} \\ &= \left[(\partial f/\partial x)^2 + (\partial f/\partial y)^2 \right]^{1/2} \end{aligned}$$

Often, this quantity is approximated by absolute values:

$$\nabla f \approx |g_x| + |g_y|$$

This approximation avoids the square and square root computations, but still behaves as a derivative (i.e., it is zero in areas of constant intensity, and has a magnitude proportional to the degree of intensity change in areas whose pixel values are variable). It is common practice to refer to the magnitude of the gradient simply as "the gradient."

A fundamental property of the gradient vector is that it points in the direction of the maximum rate of change of f at coordinates (x, y). The angle at which this maximum rate of change occurs is

$$\alpha(x, y) = \tan^{-1}\left[\frac{g_y}{g_x}\right]$$

Because g_x and g_y can be positive and/or negative independently, the arctangent must be computed using a *four-quadrant* arctangent function. MATLAB function `atan2` does this.

It is customary to approximate the derivatives by differences of gray-scale values over small neighborhoods in an image. Figure 7.26(a) shows a neighborhood of size 3×3, where the z's indicate intensity values. An approximation of the partial derivatives in the x (vertical) direction at the center point of the region is given by the difference

$$g_x = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

Similarly, the derivative in the y direction is approximated by the difference

$$g_y = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

These two quantities are easily computed at all points in an image by filtering (using function `imfilter`) the image separately with the two masks shown in Figs. 7.26(b) and (c), respectively. Then, an approximation of the corresponding gradient image is obtained by summing the absolute value of the two filtered images. The masks just discussed are the Sobel masks mentioned in Table 3.5, and thus can be generated using function `fspecial`.

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

−1	−2	−1
0	0	0
1	2	1

−1	0	1
−2	0	2
−1	0	1

a b c

FIGURE 7.26 (a) A small neighborhood. (b) and (c) Sobel masks used to compute the gradient in the x (vertical) and y (horizontal) directions, respectively, with respect to the center point of the neighborhood.

The gradient computed in the manner just described is one of the most frequently-used methods for edge detection in gray-scale images, as discussed in more detail in Chapter 11. Our interest at the moment is in computing the gradient in RGB color space. However, the method just derived is applicable in 2-D space but does not extend to higher dimensions. The only way to apply it to RGB images would be to compute the gradient of each component color image and then combine the results. Unfortunately, as we show later in this section, this is not the same as computing edges in RGB vector space directly.

The problem, then, is to define the gradient (magnitude and direction) of the vector $\mathbf{c}$ defined in Section 7.3. The following is one of the various ways in which the concept of a gradient can be extended to vector functions.

Let $\mathbf{r}$, $\mathbf{g}$, and $\mathbf{b}$ be unit vectors along the R, G, and B axes of RGB color space (see Fig. 7.2), and define the vectors

$$\mathbf{u} = \frac{\partial R}{\partial x}\mathbf{r} + \frac{\partial G}{\partial x}\mathbf{g} + \frac{\partial B}{\partial x}\mathbf{b}$$

and

$$\mathbf{v} = \frac{\partial R}{\partial y}\mathbf{r} + \frac{\partial G}{\partial y}\mathbf{g} + \frac{\partial B}{\partial y}\mathbf{b}$$

Let the quantities g_{xx}, g_{yy}, and g_{xy}, be defined in terms of the dot (inner) product of these vectors, as follows:

$$g_{xx} = \mathbf{u} \cdot \mathbf{u} = \mathbf{u}^T\mathbf{u} = \left|\frac{\partial R}{\partial x}\right|^2 + \left|\frac{\partial G}{\partial x}\right|^2 + \left|\frac{\partial B}{\partial x}\right|^2$$

$$g_{yy} = \mathbf{v} \cdot \mathbf{v} = \mathbf{v}^T\mathbf{v} = \left|\frac{\partial R}{\partial y}\right|^2 + \left|\frac{\partial G}{\partial y}\right|^2 + \left|\frac{\partial B}{\partial y}\right|^2$$

and

$$g_{xy} = \mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T\mathbf{v} = \frac{\partial R}{\partial x}\frac{\partial R}{\partial y} + \frac{\partial G}{\partial x}\frac{\partial G}{\partial y} + \frac{\partial B}{\partial x}\frac{\partial B}{\partial y}$$

Keep in mind that R, G, and B and, consequently, the g's, are functions of x and y. Using this notation, it can be shown (Di Zenzo [1986]) that the direction of maximum rate of change of $\mathbf{c}(x, y)$ as a function (x, y) is given by the angle

$$\theta(x, y) = \frac{1}{2}\tan^{-1}\left[\frac{2g_{xy}}{g_{xx} - g_{yy}}\right]$$

and that the value of the rate of change (i.e., the magnitude of the gradient) in the directions given by the elements of $\theta(x, y)$ is given by

$$F_\theta(x, y) = \left\{\frac{1}{2}\left[(g_{xx} + g_{yy}) + (g_{xx} - g_{yy})\cos 2\theta(x, y) + 2g_{xy}\sin 2\theta(x, y)\right]\right\}^{1/2}$$

Arrays $\theta(x, y)$ and $F_\theta(x, y)$ are images of the same size as the input image. The elements of $\theta(x, y)$ are the angles at each point that the gradient is calculated, and $F_\theta(x, y)$ is the gradient image.

Because $\tan(\alpha) = \tan(\alpha \pm \pi)$, if θ_0 is a solution to the preceding arctangent equation, so is $\theta_0 \pm \pi/2$. Furthermore, $F_\theta(x, y) = F_{\theta+\pi}(x, y)$, so F needs to be computed only for values of θ in the half-open interval $[0, \pi)$. The fact that the arctangent equation provides two values 90° apart means that this equation associates with each point (x, y) a pair of orthogonal directions. Along one of those directions F is maximum, and it is minimum along the other. The final result is generated by selecting the maximum at each point. The derivation of these results is rather lengthy, and we would gain little in terms of the fundamental objective of our current discussion by detailing it here. You can find the details in the paper by Di Zenzo [1986]. The partial derivatives required for implementing the preceding equations can be computed using, for example, the Sobel operators discussed earlier in this section.

The following function implements the color gradient for RGB images (see Appendix C for the code):

```
[VG, A, PPG] = colorgrad(f, T)
```

colorgrad

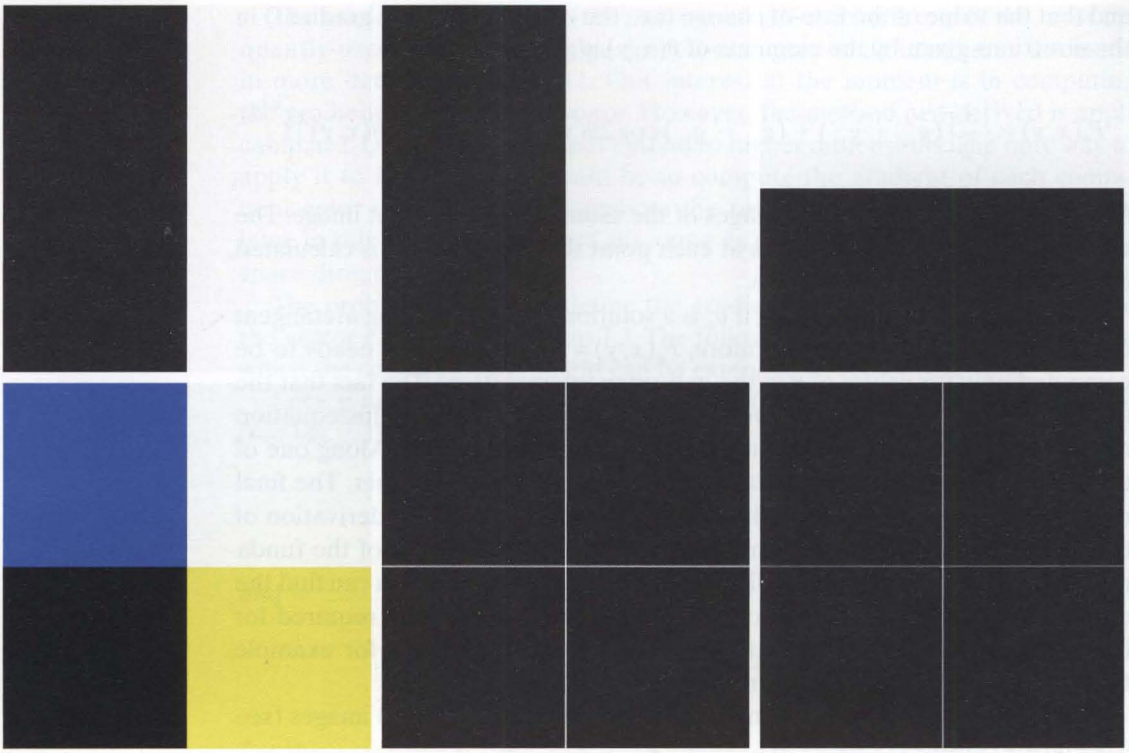
where `f` is an RGB image, `T` is an optional threshold in the range [0, 1] (the default is 0); `VG` is the RGB vector gradient $F_\theta(x, y)$; `A` is the angle image $\theta(x, y)$ in radians; and `PPG` is a gradient image formed by summing the 2-D gradient images of the *individual* color planes. All the derivatives required to implement the preceding equations are implemented in function `colorgrad` using Sobel operators. The outputs `VG` and `PPG` are normalized to the range [0, 1], and they are thresholded so that `VG(x, y) = 0` for values less than or equal to `T` and `VG(x, y) = VG(x, y)` otherwise. Similar comments apply to `PPG`.

EXAMPLE 7.12: RGB edge detection using function `colorgrad`.

■ Figures 7.27(a) through (c) show three monochrome images which, when used as RGB planes, produced the color image in Fig. 7.27(d). The objectives of this example are (1) to illustrate the use of function `colorgrad`; and (2) to show that computing the gradient of a color image by combining the gradients of its individual color planes is quite different from computing the gradient directly in RGB vector space using the method just explained.

Letting `f` represent the RGB image in Fig. 7.27(d), the command

```
>> [VG, A, PPG] = colorgrad(f);
```



a b c
d e f

FIGURE 7.27 (a) through (c) RGB component images. (d) Corresponding color image. (e) Gradient computed directly in RGB vector space. (f) Composite gradient obtained by computing the 2-D gradient of each RGB component image separately and adding the results.

produced the images VG and PPG in Figs. 7.27(e) and (f). The most important difference between these two results is how much weaker the horizontal edge in Fig. 7.27(f) is than the corresponding edge in Fig. 7.27(e). The reason is simple: The gradients of the red and green planes [Figs. 7.27(a) and (b)] produce two vertical edges, while the gradient of the blue plane yields a single horizontal edge. Adding these three gradients to form PPG produces a vertical edge with twice the intensity as the horizontal edge.

On the other hand, when the gradient of the color image is computed directly in vector space [Fig. 7.27(e)], the ratio of the values of the vertical and horizontal edges is $\sqrt{2}$ instead of 2. The reason again is simple: With reference to the color cube in Fig. 7.2(a) and the image in Fig. 7.27(d), we see that the vertical edge in the color image is between a blue and white square and a black and yellow square. The distance between these colors in the color cube is $\sqrt{2}$ but the distance between black and blue and yellow and white (the horizontal edge) is only 1. Thus, the ratio of the vertical to the horizontal differences is $\sqrt{2}$. If edge accuracy

is an issue, and especially when a threshold is used, then the difference between these two approaches can be significant. For example, if we had used a threshold of 0.6, the horizontal line in Fig. 7.27(f) would have disappeared.

When interest is mostly on edge detection with no regard for accuracy, the two approaches just discussed generally yield comparable results. For example, Figs. 7.28(b) and (c) are analogous to Figs. 7.27(e) and (f). They were obtained by applying function `colorgrad` to the image in Fig. 7.28(a). Figure 7.28(d) is the difference of the two gradient images, scaled to the range [0, 1]. The maximum absolute difference between the two images is 0.2, which translates to 51 gray levels on the familiar 8-bit range [0, 255]. However, these two gradient images are close in visual appearance, with Fig. 7.28(b) being slightly brighter in some places (for reasons similar to those explained in the previous paragraph). Thus, for this type of analysis, the simpler approach of computing the gradient of each individual component generally is acceptable. In other circumstances where accuracy is important, the vector approach is necessary. ■



a b
c d

FIGURE 7.28
(a) RGB image. (b) Gradient computed in RGB vector space. (c) Gradient computed as in Fig. 6.27(f). (d) Absolute difference between (b) and (c), scaled to the range [0, 1].

7.6.2 Image Segmentation in RGB Vector Space

Segmentation is a process that partitions an image into regions. Although segmentation is the topic of Chapter 11, we consider color region segmentation briefly here for the sake of continuity. You should have no difficulty following the discussion.

Color region segmentation using RGB color vectors is straightforward. Suppose that the objective is to segment objects of a specified color range in an RGB image. Given a set of sample color points representative of a color (or range of colors) of interest, we obtain an estimate of the "average" or "mean" color that we wish to segment. Let this average color be denoted by the RGB vector $\mathbf{m}$. The objective of segmentation is to classify each RGB pixel in a given image as having a color in the specified range or not. In order to perform this comparison, it is necessary to have a measure of similarity. One of the simplest measures is the Euclidean distance. Let $\mathbf{z}$ denote an arbitrary point in the 3-D RGB space. We say that $\mathbf{z}$ is *similar* to $\mathbf{m}$ if the distance between them is less than a specified threshold, T. The Euclidean distance between $\mathbf{z}$ and $\mathbf{m}$ is given by

We follow convention in using a superscript, T, to indicate vector or matrix transposition, and a normal, in-line, T to denote a threshold value. You can use the context in which the symbol is used to avoid confusing these unrelated uses of the same variable.

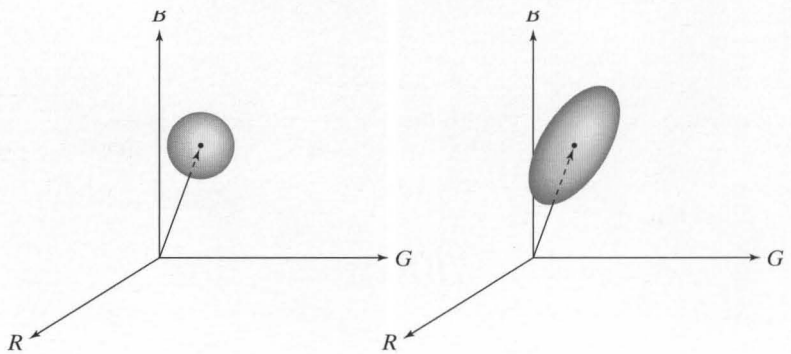
$$D(\mathbf{z}, \mathbf{m}) = \| \mathbf{z} - \mathbf{m} \| = \left[(\mathbf{z} - \mathbf{m})^T (\mathbf{z} - \mathbf{m}) \right]^{1/2}$$
$$= \left[(z_R - m_R)^2 + (z_G - m_G)^2 + (z_B - m_B)^2 \right]^{1/2}$$

where $\| \cdot \|$ is the norm of the argument, and the subscripts R, G, and B, denote the RGB components of vectors $\mathbf{z}$ and $\mathbf{m}$. The locus of points such that $D(\mathbf{z}, \mathbf{m}) \leq T$ is a solid sphere of radius T, as illustrated in Fig. 7.29(a). By definition, points contained within, or on the surface of, the sphere satisfy the specified color criterion; points outside the sphere do not. Coding these two sets of points in the image with, say, black and white, produces a binary, segmented image.

See Section 13.2 for a detailed discussion on efficient implementations for computing the Euclidean and Mahalanobis distances.

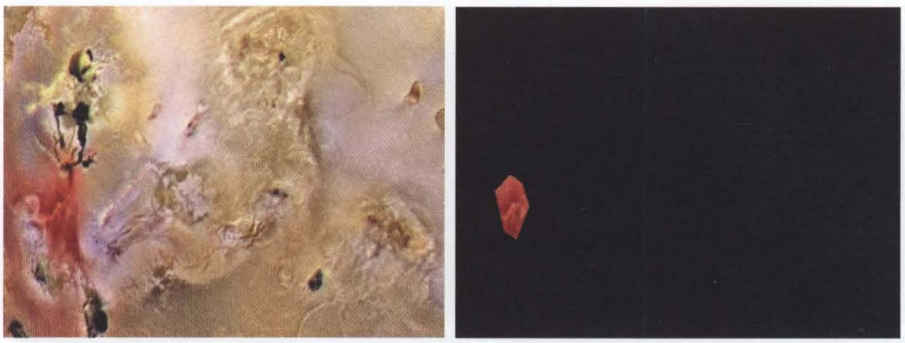
A useful generalization of the preceding equation is a distance measure of the form

$$D(\mathbf{z}, \mathbf{m}) = \left[(\mathbf{z} - \mathbf{m})^T \mathbf{C}^{-1} (\mathbf{z} - \mathbf{m}) \right]^{1/2}$$



a b

FIGURE 7.29 Two approaches for enclosing data in RGB vector space for the purpose of segmentation.



a b

FIGURE 7.30
(a) Pseudocolor of the surface of Jupiter's Moon Io. (b) Region of interest extracted interactively using function `roipoly`. (Original image courtesy of NASA.)

where $\mathbf{C}$ is the covariance matrix of the samples representative of the color we wish to segment. This distance is commonly referred to as the *Mahalanobis distance*. The locus of points such that $D(\mathbf{z}, \mathbf{m}) \leq T$ describes a solid 3-D elliptical body [see Fig. 7.29(b)] with the important property that its principal axes are oriented in the direction of maximum data spread. When $\mathbf{C} = \mathbf{I}$, the identity matrix, the Mahalanobis distance reduces to the Euclidean distance. Segmentation is as described in the preceding paragraph, except that the data are now enclosed by an ellipsoid instead of a sphere.

See Section 12.5 regarding computation of the covariance matrix and mean vector of a set of vector samples.

Segmentation in the manner just described is implemented by custom function `colorseg` (see Appendix C for the code), which has the syntax

```
S = colorseg(method, f, T, parameters)
```

`colorseg`

where `method` is either `'euclidean'` or `'mahalanobis'`, `f` is the RGB color image to be segmented, and `T` is the threshold described above. The input parameters are either `m` if `'euclidean'` is chosen, or `m` and `C` if `'mahalanobis'` is selected. Parameter `m` is the mean, $\mathbf{m}$, and `C` is the covariance matrix, $\mathbf{C}$. The output, `S`, is a two-level image (of the same size as the original) containing 0s in the points failing the threshold test, and 1s in the locations that passed the test. The 1s indicate the regions that were segmented from `f` based on color content.

EXAMPLE 7.13: RGB color image segmentation.

■ Figure 7.30(a) shows a pseudocolor image of a region on the surface of the Jupiter Moon Io. In this image, the reddish colors depict materials newly ejected from an active volcano, and the surrounding yellow materials are older sulfur deposits. This example illustrates segmentation of the reddish region using both options in function `colorseg` for comparison.

First we obtain samples representing the range of colors to be segmented. One simple way to obtain such a region of interest (ROI) is to use function `roipoly` described in Section 5.2.4 (see Example 13.2 also), which produces a binary mask of a region selected interactively. Thus, letting `f` denote the color

image in Fig. 7.30(a), the region in Fig. 7.30(b) was obtained using the commands

```
>> mask = roipoly(f); % Select region interactively.
>> red = immultiply(mask, f(:, :, 1));
>> green = immultiply(mask, f(:, :, 2));
>> blue = immultiply(mask, f(:, :, 3));
>> g = cat(3, red, green, blue);
>> figure, imshow(g);
```

where `mask` is a binary image (the same size as `f`) generated using `roipoly`.

Next, we compute the mean vector and covariance matrix of the points in the ROI, but first the coordinates of the points in the ROI must be extracted.

```
>> [M, N, K] = size(g);
>> I = reshape(g, M * N, 3);
>> idx = find(mask);
>> I = double(I(idx, 1:3));
>> [C, m] = covmatrix(I);
```

See Section 8.3.1 regarding function `reshape` and Section 12.5 regarding `covmatrix`.

The second statement rearranges the color pixels in `g` as rows of `I`, and the third statement finds the row indices of the color pixels that are not black. These are the non-background pixels of the masked image in Fig. 7.30(b).

The final preliminary computation is to determine a value for `T`. A good starting point is to let `T` be a multiple of the standard deviation of one of the color components. The main diagonal of `C` contains the variances of the RGB components, so all we have to do is extract these elements and compute their square roots:



`d = diag(C)` returns in vector `d` the main diagonal of matrix `C`.

```
>> d = diag(C);
>> sd = sqrt(d)'
    22.0643   24.2442   16.1806
```

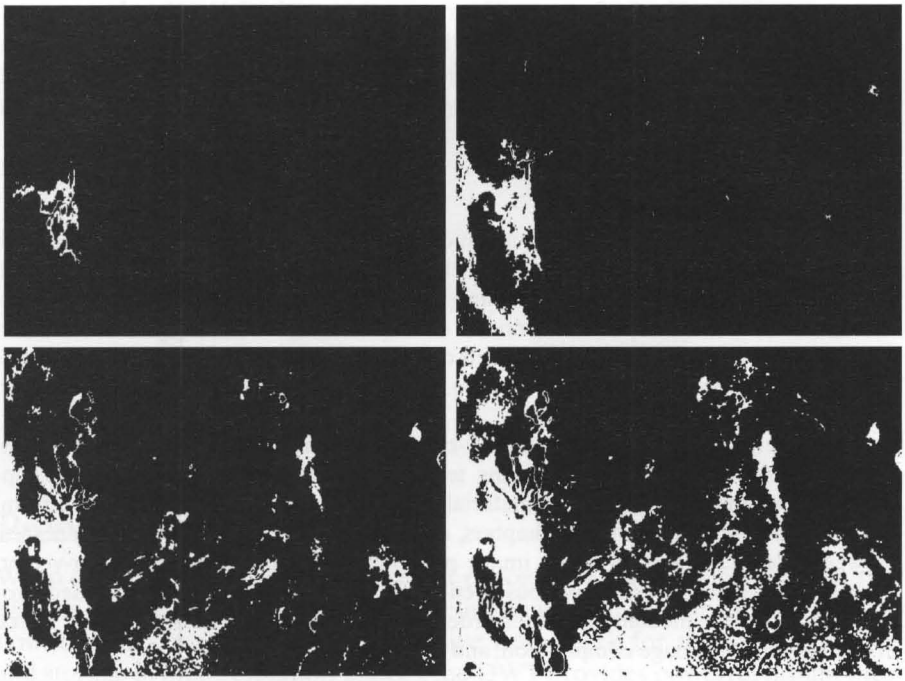
The first element of `sd` is the standard deviation of the red component of the color pixels in the ROI, and similarly for the other two components.

We now proceed to segment the image using values of `T` equal to multiples of 25, which is an approximation to the largest standard deviation: `T = 25, 50, 75, 100`. For the `'euclidean'` option with `T = 25` we use

```
>> E25 = colorseg('euclidean', f, 25, m);
```

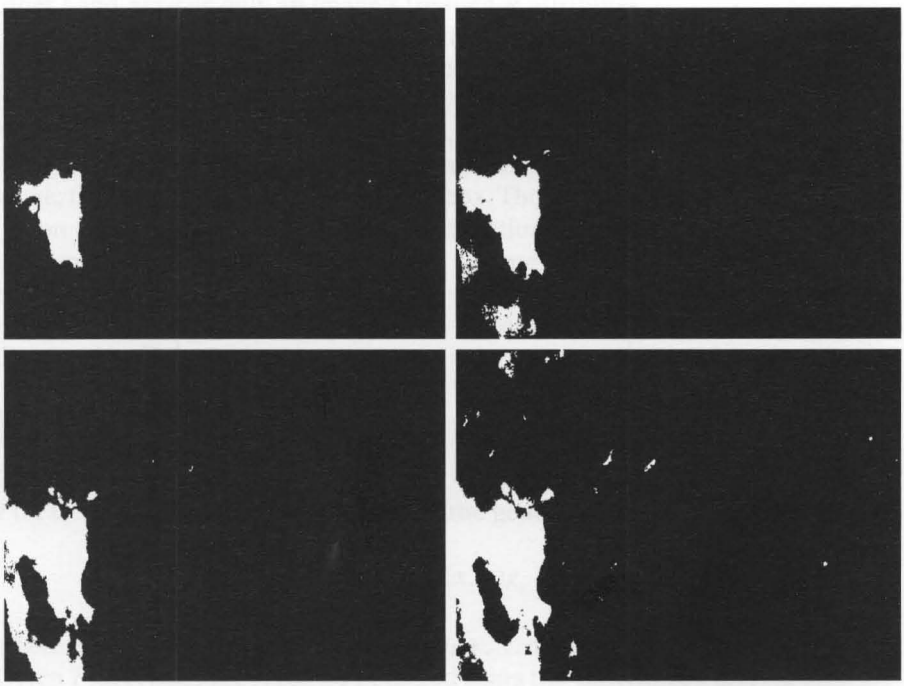
Figure 7.31(a) shows the result, and Figs. 7.31(b) through (d) show the segmentation results with `T = 50, 75, 100`. Similarly, Figs. 7.32(a) through (d) show the results obtained using the `'mahalanobis'` option with the same sequence of threshold values.

Meaningful results [depending on what we consider as *red* in Fig. 7.30(a)] were obtained with the `'euclidean'` option using `T = 25` and `50`, but `75` and `100` produced significant oversegmentation. On the other hand, the results



a b
c d

FIGURE 7.31
(a) through (d) Segmentation of Fig. 7.30(a) using option `'euclidean'` in function `colorseg` with $T = 25, 50, 75$, and 100, respectively.



a b
c d

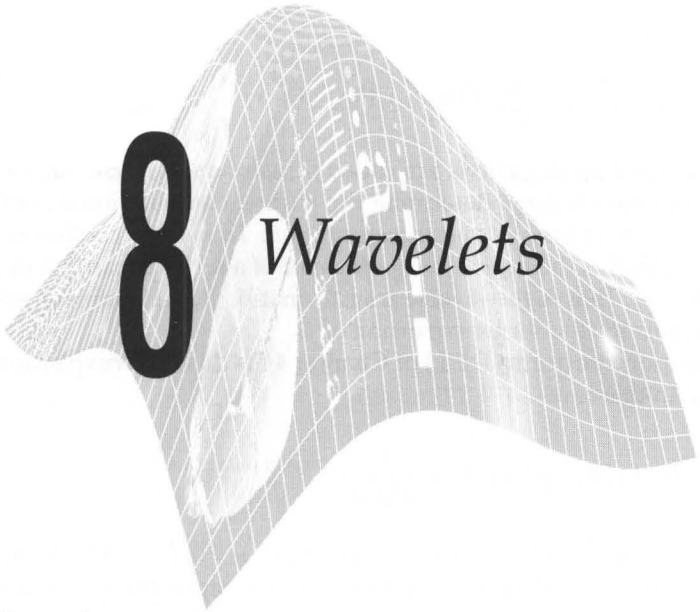
FIGURE 7.32
(a) through (d) Segmentation of Fig. 7.30(a) using option `'mahalanobis'` in function `colorseg` with $T = 25, 50, 75$, and 100, respectively. Compare with Fig. 7.31.

with the 'mahalanobis' option using the same values of T were significantly more accurate, as Fig. 7.32 shows. The reason is that the 3-D color data spread in the ROI is fitted much better in this case with an ellipsoid than with a sphere. Note that in both methods increasing T allowed weaker shades of red to be included in the segmented regions, as expected. ■

Summary

The material in this chapter is an introduction to basic topics in the application and use of color in image processing, and on the implementation of these concepts using MATLAB, Image Processing Toolbox, and new custom functions developed in the preceding sections. The area of color models is broad enough so that entire books have been written on just this topic. The models discussed here were selected for their usefulness in image processing, and also because they provide a good foundation for further study in this area.

The material on pseudocolor and full-color processing on individual color planes provides a tie to the image processing techniques developed in the previous chapters for monochrome images. The material on color vector space is a departure from the methods discussed in those chapters, and highlights some important differences between gray-scale and full-color image processing. The techniques for color-vector processing discussed in the previous section are representative of vector-based processes that include median and other order filters, adaptive and morphological filters, image restoration, image compression, and many others.



8 Wavelets

Preview

When digital images are to be viewed or processed at multiple resolutions, the *discrete wavelet transform* (DWT) is the mathematical tool of choice. In addition to being an efficient, highly intuitive framework for the representation and storage of *multiresolution* images, the DWT provides powerful insight into an image's spatial and frequency characteristics. The Fourier transform, on the other hand, reveals only an image's frequency attributes.

In this chapter, we explore both the computation and use of the discrete wavelet transform. We introduce the *Wavelet Toolbox*, a collection of MathWorks' functions designed for wavelet analysis but not included in MATLAB's Image Processing Toolbox, and develop a compatible set of routines that allow wavelet-based processing using the Image Processing Toolbox alone; that is, without the Wavelet Toolbox. These custom functions, in combination with Image Processing Toolbox functions, provide the tools needed to implement all the concepts discussed in Chapter 7 of *Digital Image Processing* by Gonzalez and Woods [2008]. They are applied in much the same way—and provide a similar range of capabilities—as toolbox functions `fft2` and `ifft2` discussed in Chapter 4.



The W on the icon is used to denote a MATLAB Wavelet Toolbox function, as opposed to a MATLAB or Image Processing Toolbox function.

8.1 Background

Consider an image $f(x, y)$ of size $M \times N$ whose forward, discrete transform, $T(u, v, \ldots)$ can be expressed in terms of the general relation

$$T(u, v, \ldots) = \sum_{x,y} f(x, y) g_{u,v,\ldots}(x, y)$$

where x and y are spatial variables and $u, v, \ldots$ are *transform domain variables*. Given $T(u, v, \ldots)$, $f(x, y)$ can be obtained using the generalized inverse discrete transform

$$f(x, y) = \sum_{u,v,\ldots} T(u, v, \ldots) h_{u,v,\ldots}(x, y)$$

The $g_{u,v,\ldots}$ and $h_{u,v,\ldots}$ in these equations are called *forward* and *inverse tranformation kernels*, respectively. They determine the nature, computational complexity, and ultimate usefulness of the *transform pair*. *Transform coefficients* $T(u, v, \ldots)$ can be viewed as the *expansion coefficients* of a series expansion of f with respect to $\{h_{u,v,\ldots}\}$. That is, the inverse transformation kernel defines a set of *expansion functions* for the series expansion of f.

The discrete Fourier transform (DFT) of Chapter 4 fits this series expansion formulation well.[†] In this case

$$h_{u,v}(x, y) = g^*_{u,v}(x, y) = \frac{1}{\sqrt{MN}} e^{j2\pi(ux/M + vy/N)}$$

where $j = \sqrt{-1}$, $*$ is the complex conjugate operator, $u = 0, 1, \ldots, M-1$ and $v = 0, 1, \ldots, N-1$. Transform domain variables u and v represent horizontal and vertical frequency, respectively. The kernels are *separable* since

$$h_{u,v}(x, y) = h_u(x) h_v(y)$$

for

$$h_u(x) = \frac{1}{\sqrt{M}} e^{j2\pi ux/M} \quad \text{and} \quad h_v(y) = \frac{1}{\sqrt{N}} e^{j2\pi vy/N}$$

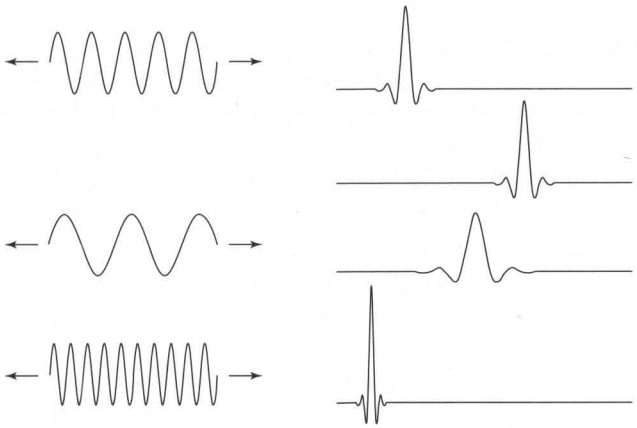
and *orthonormal* because

$$\langle h_r, h_s \rangle = \delta_{rs} = \begin{cases} 1 & r = s \\ 0 & \text{otherwise} \end{cases}$$

where $\langle\ \rangle$ is the inner product operator. The separability of the kernels simplifies the computation of the 2-D transform by allowing row-column or column-row passes of a 1-D transform to be used; orthonormality causes the forward and inverse kernels to be the complex conjugates of one another (they would be identical if the functions were real).

Unlike the discrete Fourier transform, which can be completely defined by two straightforward equations that revolve around a single pair of transformation kernels (given previously), the term *discrete wavelet transform* refers to a class of transformations that differ not only in the transformation kernels employed (and thus the expansion functions used), but also in the fundamental nature of those functions (e.g., whether they constitute an orthonormal or biorthogonal basis) and in the way in which they are applied (e.g., how many different resolutions are computed). Since the DWT encompasses a variety of unique but related transformations, we cannot write a single equation that com-

[†]In the DFT formulation of Chapter 4, a $1/MN$ term is placed in the inverse transform equation alone. Equivalently, it can be incorporated into the forward transform only, or split, as we do here, between the forward and inverse formulations as $1/\sqrt{MN}$.



a b

FIGURE 8.1
(a) The familiar Fourier expansion functions are sinusoids of varying frequency and infinite duration. (b) DWT expansion functions are "small waves" of finite duration and varying frequency.

pletely describes them all. Instead, we characterize each DWT by a transform kernel pair *or* set of parameters that defines the pair. The various transforms are related by the fact that their expansion functions are "small waves" (hence the name *wavelets*) of varying frequency and limited duration [see Fig. 8.1(b)]. In the remainder of the chapter, we introduce a number of these "small wave" kernels. Each possesses the following general properties:

Property 1: *Separability, Scalability, and Translatability*. The kernels can be represented as three separable 2-D *wavelets*

$$\psi^H(x, y) = \psi(x)\varphi(y)$$
$$\psi^V(x, y) = \varphi(x)\psi(y)$$
$$\psi^D(x, y) = \psi(x)\psi(y)$$

where $\psi^H(x, y)$, $\psi^V(x, y)$ and $\psi^D(x, y)$ are called *horizontal*, *vertical*, and *diagonal wavelets*, respectively, and one separable 2-D *scaling function*

$$\varphi(x, y) = \varphi(x)\varphi(y)$$

Each of these 2-D functions is the product of two 1-D real, square-integrable scaling and wavelet functions

$$\varphi_{j,k}(x) = 2^{j/2}\varphi(2^j x - k)$$
$$\psi_{j,k}(x) = 2^{j/2}\psi(2^j x - k)$$

Translation k determines the position of these 1-D functions along the x-axis, *scale j* determines their width—how broad or narrow they are along x—and $2^{j/2}$ controls their height or amplitude. Note that the associated expansion functions are binary scalings and integer translates of *mother* wavelet $\psi(x) = \psi_{0,0}(x)$ and scaling function $\varphi(x) = \varphi_{0,0}(x)$.

Property 2: *Multiresolution Compatibility.* The 1-D scaling function just introduced satisfies the following requirements of multiresolution analysis:

a. $\varphi_{j,k}$ is orthogonal to its integer translates.
b. The set of functions that can be represented as a series expansion of $\varphi_{j,k}$ at low scales or resolutions (i.e., small j) is contained within those that can be represented at higher scales.
c. The only function that can be represented at every scale is $f(x) = 0$.
d. Any function can be represented with arbitrary precision as $j \to \infty$.

When these conditions are met, there is a companion wavelet $\psi_{j,k}$ that, together with its integer translates and binary scalings, spans—that is, can represent—the difference between any two sets of $\varphi_{j,k}$-representable functions at adjacent scales.

Property 3: *Orthogonality.* The expansion functions [i.e., $\{\varphi_{j,k}(x)\}$] form an orthonormal or biorthogonal *basis* for the set of 1-D measurable, square-integrable functions. To be called a basis, there must be a unique set of expansion coefficients for every representable function. As was noted in the introductory remarks on Fourier kernels, $g_{u,v,\ldots} = h_{u,v,\ldots}$ for real, orthonormal kernels. For the biorthogonal case,

$$\langle h_r, g_s \rangle = \delta_{rs} = \begin{cases} 1 & r = s \\ 0 & \text{otherwise} \end{cases}$$

and g is called the *dual* of h. For a biorthogonal wavelet transform with scaling and wavelet functions $\varphi_{j,k}(x)$ and $\psi_{j,k}(x)$ the duals are denoted $\tilde{\varphi}_{j,k}(x)$ and $\tilde{\psi}_{j,k}(x)$ respectively.

8.2 The Fast Wavelet Transform

An important consequence of the above properties is that both $\varphi(x)$ and $\psi(x)$ can be expressed as linear combinations of double-resolution copies of themselves. That is, via the series expansions

$$\varphi(x) = \sum_n h_\varphi(n)\sqrt{2}\varphi(2x - n)$$

$$\psi(x) = \sum_n h_\psi(n)\sqrt{2}\varphi(2x - n)$$

where h_φ and h_ψ—the expansion coefficients—are called *scaling* and *wavelet vectors*, respectively. They are the filter coefficients of the *fast wavelet transform* (FWT), an iterative computational approach to the DWT shown in Fig. 8.2. The $W_\varphi(j, m, n)$ and $\{W_\psi^i(j, m, n)$ for $i = H, V, D\}$ outputs in this figure are the DWT coefficients at scale j. Blocks containing time-reversed scaling and wavelet vectors—the $h_\varphi(-n)$ and $h_\psi(-m)$—are *lowpass* and *highpass decomposition filters*, respectively. Finally, blocks containing a 2 and a down arrow represent *downsampling*—extracting every other point from a sequence of points. Mathematically, the series of filtering and downsampling operations used to compute $W_\psi^H(j, m, n)$ in Fig. 8.2 is, for example,

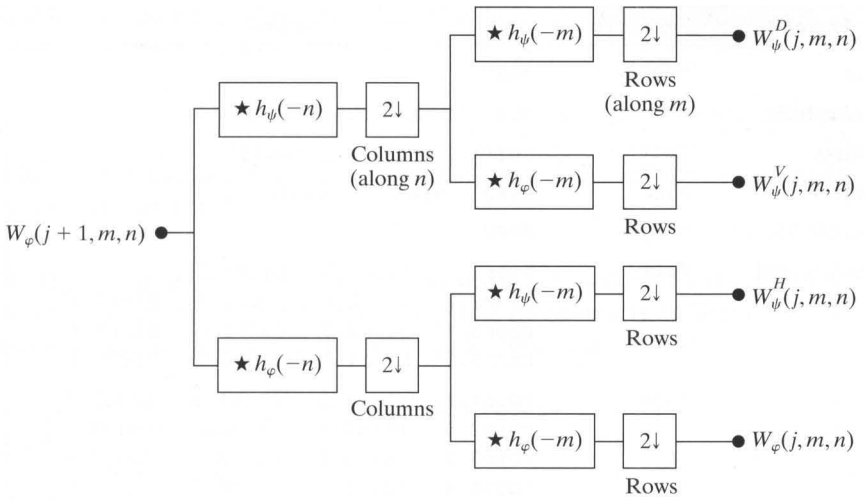


FIGURE 8.2 The 2-D fast wavelet transform (FWT) filter bank. Each pass generates one DWT scale. In the first iteration, $W_\varphi(j+1,m,n) = f(x,y)$.

$$W_\psi^H(j,m,n) = h_\psi(-m) \star \left[h_\varphi(-n) \star W_\varphi(j+1,m,n)\big|_{n=2k,\,k\geq 0}\right]\big|_{m=2k,\,k\geq 0}$$

where ★ denotes convolution. Evaluating convolutions at nonnegative, even indices is equivalent to filtering and downsampling by 2.

The input to the filter bank in Fig. 8.2 is decomposed into four lower resolution (or lower scale) components. The W_φ coefficients are created via two lowpass filters (i.e, h_φ-based) and are thus called *approximation coefficients*; $\{W_\psi^i$ for $i = H, V, D\}$ are *horizontal*, *vertical*, and *diagonal detail coefficients*, respectively. Output $W_\varphi(j,m,n)$ can be used as a subsequent input, $W_\varphi(j+1,m,n)$, to the block diagram for creating even lower resolution components; $f(x,y)$ is the highest resolution representation available and serves as the input for the first iteration. Note that the operations in Fig. 8.2 use *neither* wavelets nor scaling functions—only their associated wavelet and scaling vectors. In addition, three transform domain variables are involved—scale j and horizontal and vertical translation, n and m. These variables correspond to $u, v, \ldots$ in the first two equations of Section 8.1.

8.2.1 FWTs Using the Wavelet Toolbox

In this section, we use MATLAB's Wavelet Toolbox to compute the FWT of a simple 4×4 test image. In the next section, we will develop custom functions to do this without the Wavelet Toolbox (i.e., with the Image Processing Toolbox alone). The material here lays the groundwork for their development.

The Wavelet Toolbox provides decomposition filters for a wide variety of fast wavelet transforms. The filters associated with a specific transform are accessed via the function `wfilters`, which has the following general syntax:



Recall that the W on the icon is used to denote a MATLAB Wavelet Toolbox function, as opposed to a MATLAB or Image Processing Toolbox function.

```
[Lo_D, Hi_D, Lo_R, Hi_R] = wfilters(wname)
```

Here, input parameter `wname` determines the returned filter coefficients in accordance with Table 8.1; outputs `Lo_D`, `Hi_D`, `Lo_R`, and `Hi_R` are row vectors

TABLE 8.1 Wavelet Toolbox FWT filters and filter family names.

Wavelet	wfamily	wname
Haar	'haar'	'haar'
Daubechies	'db'	'db2', 'db3',..., 'db45'
Coiflets	'coif'	'coif1', 'coif2',..., 'coif5'
Symlets	'sym'	'sym2', 'sym3',..., 'sym45'
Discrete Meyer	'dmey'	'dmey'
Biorthogonal	'bior'	'bior1.1','bior1.3','bior1.5', 'bior2.2','bior2.4','bior2.6','bior2.8', 'bior3.1','bior3.3','bior3.5','bior3.7', 'bior3.9','bior4.4','bior5.5','bior6.8'
Reverse Biorthogonal	'rbio'	'rbio1.1','rbio1.3','rbio1.5','rbio2.2', 'rbio2.4','rbio2.6','rbio2.8','rbio3.1', 'rbio3.3','rbio3.5','rbio3.7','rbio3.9', 'rbio4.4','rbio5.5','rbio6.8'

that return the lowpass decomposition, highpass decomposition, lowpass reconstruction, and highpass reconstruction filters, respectively. (Reconstruction filters are discussed in Section 8.4.) Frequently coupled filter pairs can alternately be retrieved using

```
[F1, F2] = wfilters(wname, type)
```

with `type` set to `'d'`, `'r'`, `'l'`, or `'h'` to obtain a pair of decomposition, reconstruction, lowpass, or highpass filters, respectively. If this syntax is employed, a decomposition or lowpass filter is returned in `F1`, and its companion is placed in `F2`.

Table 8.1 lists the FWT filters included in the Wavelet Toolbox. Their properties—and other useful information on the associated scaling and wavelet functions—is available in the literature on digital filtering and multiresolution analysis. Some of the more important properties are provided by the Wavelet Toolbox's `waveinfo` and `wavefun` functions. To print a written description of wavelet family `wfamily` (see Table 8.1) on MATLAB's Command Window, for example, enter




```
waveinfo(wfamily)
```

at the MATLAB prompt. To obtain a digital approximation of an orthonormal transform's scaling and/or wavelet functions, type




```
[phi, psi, xval] = wavefun(wname, iter)
```

which returns approximation vectors, `phi` and `psi`, and evaluation vector `xval`. Positive integer `iter` determines the accuracy of the approximations by con-

trolling the number of iterations used in their computation. For biorthogonal transforms, the appropriate syntax is

```
[phi1, psi1, phi2, psi2, xval] = wavefun(wname, iter)
```

where `phi1` and `psi1` are decomposition functions and `phi2` and `psi2` are reconstruction functions.

EXAMPLE 8.1: Haar filters, scaling, and wavelet functions.

■ The oldest and simplest wavelet transform is based on the Haar scaling and wavelet functions. The decomposition and reconstruction filters for a Haar-based transform are of length 2 and can be obtained as follows:

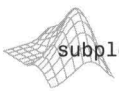
```
>> [Lo_D, Hi_D, Lo_R, Hi_R] = wfilters('haar')
Lo_D =
    0.7071    0.7071
Hi_D =
   -0.7071    0.7071
Lo_R =
    0.7071    0.7071
Hi_R =
    0.7071   -0.7071
```

Their key properties (as reported by the `waveinfo` function) and plots of the associated scaling and wavelet functions can be obtained using

```
>> waveinfo('haar');
HAARINFO Information on Haar wavelet.
    Haar Wavelet
    General characteristics: Compactly supported
    wavelet, the oldest and the simplest wavelet.
    scaling function phi = 1 on [0 1] and 0 otherwise.
    wavelet function psi = 1 on [0 0.5], = -1 on [0.5 1] and 0
    otherwise.
    Family                      Haar
    Short name                  haar
    Examples                    haar is the same as db1
    Orthogonal                  yes
    Biorthogonal                yes
    Compact support             yes
    DWT                         possible
    CWT                         possible
    Support width               1
    Filters length              2
    Regularity                  haar is not continuous
    Symmetry                    yes
    Number of vanishing
    moments for psi             1
```

```
    Reference: I. Daubechies,
    Ten lectures on wavelets,
    CBMS, SIAM, 61, 1994, 194-202.
>> [phi, psi, xval] = wavefun('haar', 10);
>> xaxis = zeros(size(xval));
>> subplot(121); plot(xval, phi, 'k', xval, xaxis, '--k');
>> axis([0 1 -1.5 1.5]); axis square;
>> title('Haar Scaling Function');
>> subplot(122); plot(xval, psi, 'k', xval, xaxis, '--k');
>> axis([0 1 -1.5 1.5]); axis square;
>> title('Haar Wavelet Function');
```

Figure 8.3 shows the display generated by the final six commands. Functions `title`, `axis`, and `plot` were described in Chapters 2 and 3; function `subplot` is used to subdivide the figure window into an array of axes or subplots. It has the following generic syntax:




```
H = subplot(m, n, p) or H = subplot(mnp)
```

where `m` and `n` are the number of rows and columns in the subplot array, respectively. Both `m` and `n` must be greater than 1. Optional output variable `H` is the handle of the subplot (i.e., axes) selected by `p`, with incremental values of `p` (beginning at 1) selecting axes along the top row of the figure window, then the second row, and so on. With or without `H`, the `p`th axes is made the current plot. Thus, the `subplot(122)` function in the commands given previously selects the plot in row 1 and column 2 of a 1×2 subplot array as the current plot; the subsequent `axis` and `title` functions then apply only to it.

The Haar scaling and wavelet functions shown in Fig. 8.3 are discontinuous and *compactly supported*, which means they are 0 outside a finite interval called the *support*. Note that the support is 1. In addition, the `waveinfo` data reveals that the Haar expansion functions are orthogonal, so that the forward and inverse transformation kernels are identical. ■

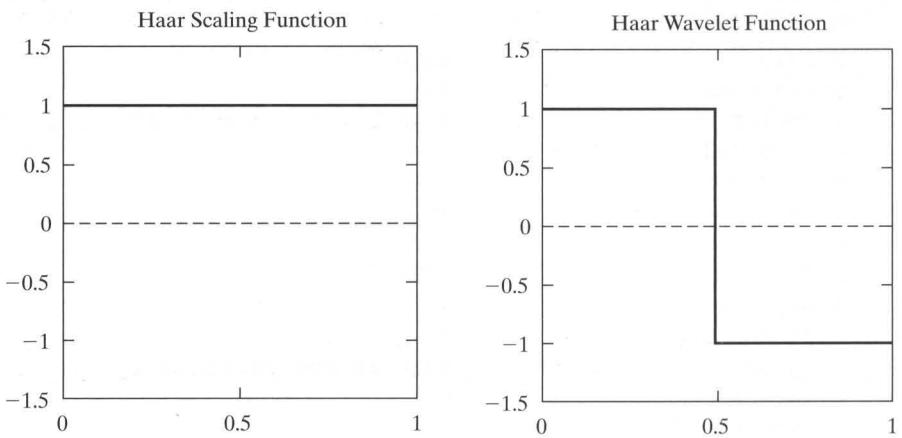


FIGURE 8.3 The Haar scaling and wavelet functions.

Given a set of decomposition filters, whether user provided or generated by the `wfilters` function, the simplest way of computing the associated wavelet transform is through the Wavelet Toolbox's `wavedec2` function. It is invoked using

```
[C, S] = wavedec2(X, N, Lo_D, Hi_D)
```



where `X` is a 2-D image or matrix, `N` is the number of scales to be computed (i.e., the number of passes through the FWT filter bank in Fig. 8.2), and `Lo_D` and `Hi_D` are decomposition filters. The slightly more efficient syntax

```
[C, S] = wavedec2(X, N, wname)
```

in which `wname` assumes a value from Table 8.1, can also be used. Output data structure `[C, S]` is composed of row vector `C` (class `double`), which contains the computed wavelet transform coefficients, and bookkeeping matrix `S` (also class `double`), which defines the arrangement of the coefficients in `C`. The relationship between `C` and `S` is introduced in the next example and described in detail in Section 8.3.

EXAMPLE 8.2: A simple FWT using Haar filters.

■ Consider the following single-scale wavelet transform with respect to Haar wavelets:

```
>> f = magic(4)
f =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> [c1, s1] = wavedec2(f, 1, 'haar')
c1 =
  Columns 1 through 9
    17.0000   17.0000   17.0000   17.0000    1.0000
    -1.0000   -1.0000    1.0000    4.0000
  Columns 10 through 16
    -4.0000   -4.0000    4.0000   10.0000    6.0000
    -6.0000  -10.0000
s1 =
     2     2
     2     2
     4     4
```

Here, a 4×4 magic square `f` is transformed into a 1×16 wavelet decomposition vector `c1` and 3×2 bookkeeping matrix `s1`. The entire transformation

is performed with a single execution (with `f` used as the input) of the operations depicted in Fig. 8.2. Four 2×2 outputs—a downsampled approximation and three directional (horizontal, vertical, and diagonal) detail matrices—are generated. Function `wavedec2` concatenates these 2×2 matrices columnwise in row vector `c1` beginning with the approximation coefficients and continuing with the horizontal, vertical, and diagonal details. That is, `c1(1)` through `c1(4)` are approximation coefficients $W_\varphi(1,0,0)$, $W_\varphi(1,1,0)$, $W_\varphi(1,0,1)$, and $W_\varphi(1,1,1)$ from Fig. 8.2 with the scale of `f` assumed arbitrarily to be 2; `c1(5)` through `c1(8)` are $W_\psi^H(1,0,0)$, $W_\psi^H(1,1,0)$, $W_\psi^H(1,0,1)$, and $W_\psi^H(1,1,1)$; and so on. It we were to extract the horizontal detail coefficient matrix from vector `c1`, for example, we would get

$$W_\psi^H = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Bookkeeping matrix `s1` provides the sizes of the matrices that have been concatenated a column at a time into row vector `c1`—plus the size of the original image `f` [in vector `s1(end, :)`]. Vectors `s1(1, :)` and `s1(2, :)` contain the sizes of the computed approximation matrix and three detail coefficient matrices, respectively. The first element of each vector is the number of rows in the referenced detail or approximation matrix; the second element is the number of columns.

When the single-scale transform described above is extended to two scales, we get

```
>> [c2, s2] = wavedec2(f, 2, 'haar')
c2 =
    Columns 1 through 9
        34.0000      0            0          0.0000      1.0000
        -t1.0000     -1.0000      1.0000     4.0000

    Columns 10 through 16
        -4.0000     -4.0000      4.0000    10.0000      6.0000
        -6.0000    -10.0000

s2 =
    1      1
    1      1
    2      2
    4      4
```

Note that `c2(5:16) = c1(5:16)`. Elements `c1(1:4)`, which were the approximation coefficients of the single-scale transform, have been fed into the filter bank of Fig. 8.2 to produce four 1×1 outputs: $W_\varphi(0,0,0)$, $W_\psi^H(0,0,0)$, $W_\psi^V(0,0,0)$, and $W_\psi^D(0,0,0)$. These outputs are concatenated columnwise (though they are 1×1 matrices here) in the same order that was used in the preceding single-scale transform and substituted for the approximation coefficients from which they were derived. Bookkeeping matrix `s2` is then updated to reflect the fact that the single 2×2 approximation matrix in `c1` has

been replaced by four 1×1 detail and approximation matrices in `c2`. Thus, `s2(end, :)` is once again the size of the original image, `s2(3, :)` is the size of the three detail coefficient matrices at scale 1, `s2(2, :)` is the size of the three detail coefficient matrices at scale 0, and `s2(1, :)` is the size of the final approximation. ■

To conclude this section, we note that because the FWT is based on digital filtering techniques and thus convolution, border distortions can arise. To minimize these distortions, the border must be treated differently from the other parts of the image. When filter elements fall outside the image during the convolution process, values must be assumed for the area, which is about the size of the filter, outside the image. Many Wavelet Toolbox functions, including the `wavedec2` function, extend or pad the image being processed based on global parameter `dwtmode`. To examine the active extension mode, enter `st = dwtmode('status')` or simply `dwtmode` at the MATLAB command prompt (e.g., `> > dwtmode`). To set the extension mode to `STATUS`, enter `dwtmode(STATUS);` to make `STATUS` the default extension mode, use `dwtmode('save', STATUS)`. The supported extension modes and corresponding `STATUS` values are listed in Table 8.2.



8.2.2 FWTs without the Wavelet Toolbox

In this section, we develop a pair of custom functions, `wavefilter` and `wavefast`, to replace the Wavelet Toolbox functions, `wfilters` and `wavedec2`, of the previous section. Our goal is to provide additional insight into the mechanics of computing FWTs, and to begin the process of building a "standalone package" for wavelet-based image processing without the Wavelet Toolbox. This process is completed in Sections 8.3 and 8.4, and the resulting set of functions is used to generate the examples in Section 8.5.

The first step is to devise a function for generating wavelet decomposition and reconstruction filters. The following function, which we call `wavefilter`, uses a standard `switch` construct, together with `case` and `otherwise`, to do

STATUS	Description
`'sym'`	The image is extended by mirror reflecting it across its borders. This is the normal default mode.
`'zpd'`	The image is extended by padding with a value of 0.
`'spd', 'sp1'`	The image is extended by first-order derivative extrapolation—or padding with a linear extension of the outmost two border values.
`'sp0'`	The image is extended by extrapolating the border values—that is, by boundary value replication.
`'ppd'`	The image is extended by periodic padding.
`'per'`	The image is extended by periodic padding after it has been padded (if necessary) to an even size using `'sp0'` extension.

TABLE 8.2 Wavelet Toolbox image extension or padding modes.

this in a readily extendable manner. Although `wavefilter` provides only the filters examined in Chapters 7 and 8 of *Digital Image Processing* (Gonzalez and Woods [2008]), other wavelet transforms can be accommodated by adding (as new "cases") the appropriate decomposition and reconstruction filters from the literature.

`wavefilter`

```
function [varargout] = wavefilter(wname, type)
%WAVEFILTER Create wavelet decomposition and reconstruction filters.
%   [VARARGOUT] = WAVEFILTER(WNAME, TYPE) returns the decomposition
%   and/or reconstruction filters used in the computation of the
%   forward and inverse FWT (fast wavelet transform).
%
%   EXAMPLES:
%     [ld, hd, lr, hr] = wavefilter('haar') Get the low and highpass
%                                           decomposition (ld, hd)
%                                           and reconstruction
%                                           (lr, hr) filters for
%                                           wavelet 'haar'.
%     [ld, hd] = wavefilter('haar','d')     Get decomposition filters
%                                           ld and hd.
%     [lr, hr] = wavefilter('haar','r')     Get reconstruction
%                                           filters lr and hr.
%
%   INPUTS:
%     WNAME             Wavelet Name
%     ------------------------------------------------------
%     'haar' or 'db1'   Haar
%     'db4'             4th order Daubechies
%     'sym4'            4th order Symlets
%     'bior6.8'         Cohen-Daubechies-Feauveau biorthogonal
%     'jpeg9.7'         Antonini-Barlaud-Mathieu-Daubechies
%
%     TYPE              Filter Type
%     ------------------------------------------------------
%     'd'               Decomposition filters
%     'r'               Reconstruction filters
%
%   See also WAVEFAST and WAVEBACK.

% Check the input and output arguments.
error(nargchk(1, 2, nargin));

if (nargin == 1 && nargout ~= 4) || (nargin == 2 && nargout ~= 2)
   error('Invalid number of output arguments.');
end

if nargin == 1 && ~ischar(wname)
   error('WNAME must be a string.');
end
```

```
if nargin == 2 && ~ischar(type)
   error('TYPE must be a string.');
end

% Create filters for the requested wavelet.
switch lower(wname)
case {'haar', 'db1'}
   ld = [1 1]/sqrt(2);      hd = [-1 1]/sqrt(2);
   lr = ld;                 hr = -hd;

case 'db4'
   ld = [-1.059740178499728e-002 3.288301166698295e-002 ...
         3.084138183598697e-002 -1.870348117188811e-001 ...
         -2.798376941698385e-002 6.308807679295904e-001 ...
         7.148465705525415e-001 2.303778133088552e-001];
   t = (0:7);
   hd = ld;    hd(end:-1:1) = cos(pi * t) .* ld;
   lr = ld;    lr(end:-1:1) = ld;
   hr = cos(pi * t) .* ld;

case 'sym4'
   ld = [-7.576571478927333e-002 -2.963552764599851e-002 ...
         4.976186676320155e-001 8.037387518059161e-001 ...
         2.978577956052774e-001 -9.921954357684722e-002 ...
         -1.260396726203783e-002 3.222310060404270e-002];
   t = (0:7);
   hd = ld;    hd(end:-1:1) = cos(pi * t) .* ld;
   lr = ld;    lr(end:-1:1) = ld;
   hr = cos(pi * t) .* ld;

case 'bior6.8'
   ld = [0 1.908831736481291e-003 -1.914286129088767e-003 ...
         -1.699063986760234e-002 1.193456527972926e-002 ...
         4.973290349094079e-002 -7.726317316720414e-002 ...
         -9.405920349573646e-002 4.207962846098268e-001 ...
         8.259229974584023e-001 4.207962846098268e-001 ...
         -9.405920349573646e-002 -7.726317316720414e-002 ...
         4.973290349094079e-002 1.193456527972926e-002 ...
         -1.699063986760234e-002 -1.914286129088767e-003 ...
         1.908831736481291e-003];
   hd = [0 0 0 1.442628250562444e-002 -1.446750489679015e-002 ...
         -7.872200106262882e-002 4.036797903033992e-002 ...
         4.178491091502746e-001 -7.589077294536542e-001 ...
         4.178491091502746e-001 4.036797903033992e-002 ...
         -7.872200106262882e-002 -1.446750489679015e-002 ...
         1.442628250562444e-002 0 0 0 0];
   t = (0:17);
   lr = cos(pi * (t + 1)) .* hd;
   hr = cos(pi * t) .* ld;
```

```
case 'jpeg9.7'
   ld = [0 0.02674875741080976 -0.01686411844287495 ...
         -0.07822326652898785 0.2668641184428723 ...
         0.6029490182363579 0.2668641184428723 ...
         -0.07822326652898785 -0.01686411844287495 ...
         0.02674875741080976];
   hd = [0 -0.09127176311424948 0.05754352622849957 ...
         0.5912717631142470 -1.115087052456994 ...
         0.5912717631142470 0.05754352622849957 ...
         -0.09127176311424948 0 0];
   t = (0:9);
   lr = cos(pi * (t + 1)) .* hd;
   hr = cos(pi * t) .* ld;

otherwise
   error('Unrecognizable wavelet name (WNAME).');
end

% Output the requested filters.
if (nargin == 1)
   varargout(1:4) = {ld, hd, lr, hr};
else
   switch lower(type(1))
   case 'd'
      varargout = {ld, hd};
   case 'r'
      varargout = {lr, hr};
   otherwise
      error('Unrecognizable filter TYPE.');
   end
end
```

Note that for each orthonormal filter in `wavefilter` (i.e., `'haar'`, `'db4'`, and `'sym4'`), the reconstruction filters are time-reversed versions of the decomposition filters and the highpass decomposition filter is a modulated version of its lowpass counterpart. Only the lowpass decomposition filter coefficients need to be explicitly enumerated in the code. The remaining filter coefficients can be computed from them. In `wavefilter`, time reversal is carried out by reordering filter vector elements from last to first with statements like `lr(end:-1:1) = ld`. Modulation is accomplished by multiplying the components of a known filter by `cos(pi*t)`, which alternates between 1 and –1 as `t` increases from 0 in integer steps. For each biorthogonal filter in `wavefilter` (i.e., `'bior6.8'` and `'jpeg9.7'`), both the lowpass and highpass decomposition filters are specified; the reconstruction filters are computed as modulations of them. Finally, we note that the filters generated by `wavefilter` are of even length. Moreover, zero padding is used to ensure that the lengths of the decomposition and reconstruction filters of each wavelet are identical.

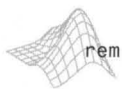
Given a pair of `wavefilter` generated decomposition filters, it is easy to write a general-purpose routine for the computation of the related FWT. The

goal is to devise an efficient algorithm based on the filtering and downsampling operations in Fig. 8.2. To maintain compatibility with the existing Wavelet Toolbox, we employ the same decomposition structure (i.e., `[C, S]` where `C` is a decomposition vector and `S` is a bookkeeping matrix). Because `wavedec2` can accept $M \times N \times 3$ inputs, we also accept arrays that are *extended* along a third dimension. That is, the input can contain more than one 2-D array—like the red, green, and blue components of an RGB image. Each 2-D array of the extended array is called a *page* and its third index is called the *page index*. The following routine, which we call `wavefast`, uses symmetric image extension to reduce the border distortion associated with the computed FWT(s):

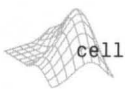
wavefast

```
function [c, s] = wavefast(x, n, varargin)
%WAVEFAST Computes the FWT of a '3-D extended' 2-D array.
%   [C, L] = WAVEFAST(X, N, LP, HP) computes 'PAGES' 2D N-level
%   FWTs of a 'ROWS x COLUMNS x PAGES' matrix X with respect to
%   decomposition filters LP and HP.
%
%   [C, L] = WAVEFAST(X, N, WNAME) performs the same operation but
%   fetches filters LP and HP for wavelet WNAME using WAVEFILTER.
%
%   Scale parameter N must be less than or equal to log2 of the
%   maximum image dimension.  Filters LP and HP must be even. To
%   reduce border distortion, X is symmetrically extended. That is,
%   if X = [c1 c2 c3 ... cn] (in 1D), then its symmetric extension
%   would be [... c3 c2 c1 c1 c2 c3 ... cn cn cn-1 cn-2 ...].
%
%   OUTPUTS:
%     Vector C is a coefficient decomposition vector:
%
%      C = [ a1(n)...ak(n) h1(n)...hk(n) v1(n)...vk(n)
%            d1(n)...dk(n) h1(n-1)... d1(1)...dk(1) ]
%
%     where ai, hi, vi, and di for i = 0,1,...k are columnwise
%     vectors containing approximation, horizontal, vertical, and
%     diagonal coefficient matrices, respectively, and k is the
%     number of pages in the 3-D extended array X. C has 3n + 1
%     sections where n is the number of wavelet decompositions.
%
%     Matrix S is an [(n+2) x 2] bookkeeping matrix if k = 1;
%     else it is [(n+2) x 3]:
%
%      S = [ sa(n, :); sd(n, :); sd(n-1, :); ... ; sd(1, :); sx ]
%
%     where sa and sd are approximation and detail size entries.
%
%   See also WAVEBACK and WAVEFILTER.

% Check the input arguments for reasonableness.
error(nargchk(3, 4, nargin));
```

```
if nargin == 3
   if ischar(varargin{1})
      [lp, hp] = wavefilter(varargin{1}, 'd');
   else
      error('Missing wavelet name.');
   end
else
   lp = varargin{1};   hp = varargin{2};
end

% Get the filter length, 'lp', input array size, 'sx', and number of
% pages, 'pages', in extended 2-D array x.
fl = length(lp);       sx = size(x);        pages = size(x, 3);

if ((ndims(x) ~= 2) && (ndims(x) ~= 3)) || (min(sx) < 2) ...
        || ~isreal(x) || ~isnumeric(x)
   error('X must be a real, numeric 2-D or 3-D matrix.');
end

if (ndims(lp) ~= 2) || ~isreal(lp) || ~isnumeric(lp) ...
       || (ndims(hp) ~= 2) || ~isreal(hp) || ~isnumeric(hp) ...
       || (fl ~= length(hp)) || rem(fl, 2) ~= 0
   error(['LP and HP must be even and equal length real, ' ...
          'numeric filter vectors.']);
end

if ~isreal(n) || ~isnumeric(n) || (n < 1) || (n > log2(max(sx)))
   error(['N must be a real scalar between 1 and ' ...
          'log2(max(size((X)))).']);
end

% Init the starting output data structures and initial approximation.
c = [];        s = sx(1:2);
app = cell(pages, 1);
for i = 1:pages
   app{i} = double(x(:, :, i));
end

% For each decomposition ...
for i = 1:n
   % Extend the approximation symmetrically.
   [app, keep] = symextend(app, fl, pages);

   % Convolve rows with HP and downsample. Then convolve columns
   % with HP and LP to get the diagonal and vertical coefficients.
   rows = symconv(app, hp, 'row', fl, keep, pages);
   coefs = symconv(rows, hp, 'col', fl, keep, pages);
   c = addcoefs(c, coefs, pages);
   s = [size(coefs{1}); s];
   coefs = symconv(rows, lp, 'col', fl, keep, pages);
   c = addcoefs(c, coefs, pages);
```



`rem (X, Y)` returns the remainder of the division of `X` by `Y`.



`cell (m, n)` creates an `m` by `n` array of empty matrices.

```
   % Convolve rows with LP and downsample. Then convolve columns
   % with HP and LP to get the horizontal and next approximation
   % coeffcients.
   rows = symconv(app, lp, 'row', fl, keep, pages);
   coefs = symconv(rows, hp, 'col', fl, keep, pages);
   c = addcoefs(c, coefs, pages);
   app = symconv(rows, lp, 'col', fl, keep, pages);
end

% Append the final approximation structures.
c = addcoefs(c, app, pages);
s = [size(app{1}); s];
if ndims(x) > 2
   s(:, 3) = size(x, 3);
end

%-----------------------------------------------------------------------%
function nc = addcoefs(c, x, pages)
% Add 'pages' array coefficients to the wavelet decomposition vector.

nc = c;
for i = pages:-1:1
   nc = [x{i}(:)' nc];
end

%-----------------------------------------------------------------------%
function [y, keep] = symextend(x, fl, pages)
% Compute the number of coefficients to keep after convolution and
% downsampling. Then extend the 'pages' arrays of x in both
% dimensions.

y = cell(pages, 1);
for i = 1:pages
   keep = floor((fl + size(x{i}) - 1) / 2);
   y{i} = padarray(x{i}, [(fl - 1) (fl - 1)], 'symmetric', 'both');
end

%-----------------------------------------------------------------------%
function y = symconv(x, h, type, fl, keep, pages)
% For the 'pages' 2-D arrays in x, convolve the rows or columns with
% h, downsample, and extract the center section since symmetrically
% extended.

y = cell(pages, 1);
for i = 1:pages
   if strcmp(type, 'row')
       y{i} = conv2(x{i}, h);
       y{i} = y{i}(:, 1:2:end);
       y{i} = y{i}(:, fl / 2 + 1:fl / 2 + keep(2));
   else
```



`C = conv2 (A, B)` performs the 2-D convolution of matrices A and B.

```
        y{i} = conv2(x{i}, h');
        y{i} = y{i}(1:2:end, :);
        y{i} = y{i}(fl / 2 + 1:fl / 2 + keep(1), :);
    end
end
```

As can be seen in the main routine, only one `for` loop, which cycles through the decomposition levels (or scales) that are generated, is used to orchestrate the entire forward transform computation. For each execution of the loop, the current approximation cell array `app`–whose elements are initially set to the 2-D images (or pages) of `x`—are symmetrically extended by internal function `symextend`. This function calls `padarray`, which was introduced in Section 3.4.2, to extend the matrices of `app` in two dimensions by mirror reflecting `fl - 1` of their elements (the length of the decomposition filter minus 1) across their borders.

If `x` is a 2-D array, there is only one element in `app`. If `x` is a 3-D array, its third index determines the number of 2-D arrays (or pages) that are to be transformed. In either case, the first and second indices determine the size of the 2-D array or arrays in `app`.

Function `symextend` returns a cell array of extended approximation matrices and the number of pixels that should be extracted from the center of any subsequently convolved and downsampled results. The rows of the extended approximations are next convolved with highpass decomposition filter `hp` and downsampled via `symconv`. This function is described in the following paragraph. Convolved output, `rows` (also a cell array), is then submitted to `symconv` to convolve and downsample its columns with filters `hp` and `lp`—generating the diagonal and vertical detail coefficients of the top two branches of Fig. 8.2. These results are inserted into decomposition vector `c` by function `addcoefs` (working from the last element toward the first) and the process is repeated in accordance with Fig. 8.2 to generate the horizontal detail and approximation coefficients (the bottom two branches of the figure).

Function `symconv` uses the `conv2` function to do the bulk of the transform computation work. It convolves filter `h` with the rows or columns of each matrix in cell array `x` (depending on `type`), discards the even indexed rows or columns (i.e., downsamples by 2), and extracts the center `keep` elements of each row or column. Invoking `conv2` with cell array `x` and row filter vector `h` initiates a row-by-row convolution with each matrix in `x`; using column filter vector `h'` results in columnwise convolutions.

EXAMPLE 8.3: Comparing the execution times of `wavefast` and `wavedec2`.

■ The following test routine uses function `timeit` from Chapter 2 to compare the execution times of the Wavelet Toolbox function `wavedec2` and custom function `wavefast`:

```
function [ratio, maxdiff] = fwtcompare(f, n, wname)
%FWTCOMPARE Compare wavedec2 and wavefast.
%   [RATIO, MAXDIFF] = FWTCOMPARE(F, N, WNAME) compares the
%   operation of Wavelet Toolbox function WAVEDEC2 and custom
%   function WAVEFAST.
%
%   INPUTS:
%     F           Image to be transformed.
```

```
%     N           Number of scales to compute.
%     WNAME       Wavelet to use.
%
%   OUTPUTS:
%     RATIO       Execution time ratio (custom/toolbox)
%     MAXDIFF     Maximum coefficient difference.

% Get transform and computation time for wavedec2.
w1 = @() wavedec2(f, n, wname);
reftime = timeit(w1);

% Get transform and computation time for wavefast.
w2 = @() wavefast(f, n, wname);
t2 = timeit(w2);

% Compare the results.
ratio = t2 / reftime;
maxdiff = abs(max(w1() - w2()));
```

For the image of Fig. 8.4 and a five-scale wavelet transform with respect to 4th order Daubechies' wavelets, `fwtcompare` yields

```
>> f = imread('vase.tif');
>> [ratio, maxdifference] = fwtcompare(f, 5, 'db4')
ratio =
    0.7303
maxdifference =
    3.2969e-012
```

Note that custom function `wavefast` was faster than its Wavelet Toolbox counterpart while producing virtually identical results. ■

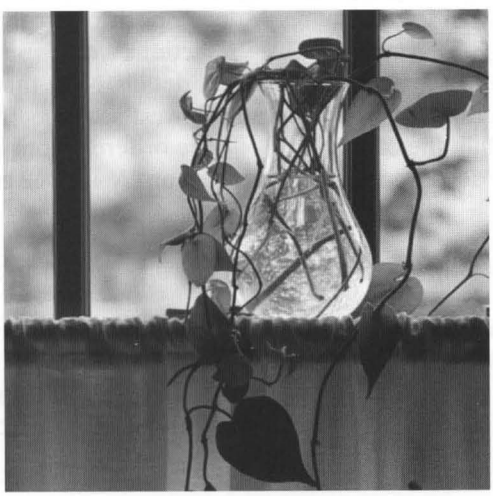


FIGURE 8.4
A 512 × 512 image of a vase.

8.3 Working with Wavelet Decomposition Structures

The wavelet transformation functions of the previous two sections generate *nondisplayable* data structures of the form {**c**, **S**}, where **c** is a transform coefficient vector and **S** is a bookkeeping matrix that defines the arrangement of coefficients in **c**. To process images, we must be able to examine and/or modify **c**. In this section, we formally define {**c**, **S**}, examine some of the Wavelet Toolbox functions for manipulating it, and develop a set of custom functions that can be used without the Wavelet Toolbox. These functions are then used to build a general purpose routine for displaying **c**.

The representation scheme introduced in Example 8.2 integrates the coefficients of a multiscale two-dimensional wavelet transform into a single, one-dimensional vector

$$\mathbf{c} = [\mathbf{A}_N(:)' \quad \mathbf{H}_N(:)' \quad \ldots \quad \mathbf{H}_i(:)' \quad \mathbf{V}_i(:)' \quad \mathbf{D}_i(:)' \quad \cdots \quad \mathbf{V}_1(:)' \quad \mathbf{D}_1(:)']$$

where $\mathbf{A}_N$ is the approximation coefficient matrix of the Nth decomposition level and $\mathbf{H}_i$, $\mathbf{V}_i$, and $\mathbf{D}_i$ for $i = 1, 2, \ldots N$ are the horizontal, vertical, and diagonal transform coefficient matrices for level i. Here, $\mathbf{H}_i(:)'$ for example, is the row vector formed by concatenating the transposed columns of matrix $\mathbf{H}_i$. That is, if

$$\mathbf{H}_i = \begin{bmatrix} 3 & -2 \\ 1 & 6 \end{bmatrix}$$

then

$$\mathbf{H}_i(:) = \begin{bmatrix} 3 \\ 1 \\ -2 \\ 6 \end{bmatrix} \quad \text{and} \quad \mathbf{H}_i(:)' = [3 \quad 1 \quad -2 \quad 6]$$

Because the equation for **c** assumes N decompositions (or passes through the filter bank in Fig. 8.2), **c** contains $3N + 1$ sections—one approximation and N groups of horizontal, vertical, and diagonal details. Note that the highest scale coefficients are computed when $i = 1$; the lowest scale coefficients are associated with $i = N$. Thus, the coefficients of **c** are ordered from low to high scale.

Matrix **S** of the decomposition structure is an $(N + 2) \times 2$ bookkeeping array of the form

$$\mathbf{S} = [\mathbf{sa}_N; \quad \mathbf{sd}_N; \quad \mathbf{sd}_{N-1}; \quad \ldots \quad \mathbf{sd}_i; \quad \ldots \quad \mathbf{sd}_1; \quad \mathbf{sf}]$$

where $\mathbf{sa}_N$, $\mathbf{sd}_i$, and **sf** are 1×2 vectors containing the horizontal and vertical dimensions of Nth-level approximation $\mathbf{A}_N$, ith-level details ($\mathbf{H}_i$, $\mathbf{V}_i$, and $\mathbf{D}_i$ for $i = 1, 2, \ldots N$), and original image **F**, respectively. The information in **S** can be used to locate the individual approximation and detail coefficient matrices of **c**.

Note that the semicolons in the preceding equation indicate that the elements of **S** are organized as a column vector.

When a 3-D array is transformed, it is treated as an *extended 2-D array*—a "book" of 2-D arrays in which the number of "pages" is determined by the third index of the 3-D array being transformed. An extended array might contain the color components of a full-color image (see the RGB color planes in Fig. 7.1) or the individual frames that make up a time sequence of images. To compute the FWT of a 3-D array, each 2-D array or page is transformed independently, with the resulting decomposition coefficients interleaved in a single {**c**,**S**} structure. The elements of vector **c** become

You can also think of it as a "stack" of 2-D arrays in which the number of "stacked arrays" is determined by the third array index.

$$\begin{aligned}
\mathbf{A}_N(:)' &= [\mathbf{A}_N^1(:)' \quad \mathbf{A}_N^2(:)' \quad \ldots \quad \mathbf{A}_N^K(:)'] \\
\mathbf{H}_i(:)' &= [\mathbf{H}_i^1(:)' \quad \mathbf{H}_i^2(:)' \quad \ldots \quad \mathbf{H}_i^K(:)'] \\
\mathbf{V}_i(:)' &= [\mathbf{V}_i^1(:)' \quad \mathbf{V}_i^2(:)' \quad \ldots \quad \mathbf{V}_i^K(:)'] \\
\mathbf{D}_i(:)' &= [\mathbf{D}_i^1(:)' \quad \mathbf{D}_i^2(:)' \quad \ldots \quad \mathbf{D}_i^K(:)']
\end{aligned}$$

where K is the number of pages (or 2-D arrays) in the extended array, i is the decomposition level, and the superscripts on **A**, **H**, **V**, and **D** designate the pages from which the associated FWT coefficients are derived. Thus, the approximation and detail coefficients of all pages are concatenated at each decomposition level. As before, **c** is composed of $3N+1$ sections, but bookkeeping matrix **S** becomes an $(N+2)\times 3$ array in which the third column specifies the number of 2-D arrays in **c**.

EXAMPLE 8.4: Wavelet Toolbox functions for manipulating transform decomposition vector **c**.

■ The Wavelet Toolbox provides a variety of functions for locating, extracting, reformatting, and/or manipulating the approximation and horizontal, vertical, and diagonal coefficients of **c** as a function of decomposition level. We introduce them here to illustrate the concepts just discussed and to prepare the way for the alternative functions that will be developed in the next section. Consider, for example, the following sequence of commands:

```
>> f = magic(8);
>> [c1, s1] = wavedec2(f, 3, 'haar');
>> size(c1)
ans =
     1    64
>> s1
s1 =
     1     1
     1     1
     2     2
     4     4
     8     8
>> approx = appcoef2(c1, s1, 'haar')
approx =
   260.0000
```

```
>> horizdet2 = detcoef2('h', c1, s1, 2)
horizdet2 =
    1.0e-013 *
            0   -0.2842
            0         0
>> newc1 = wthcoef2('h', c1, s1, 2);
>> newhorizdet2 = detcoef2('h', newc1, s1, 2)
newhorizdet2 =
    0     0
    0     0
```

Here, K is 1 and a three-level decomposition with respect to Haar wavelets is performed on a single 8×8 magic square using the `wavedec` function. The resulting coefficient vector, `c1`, is of size 1×64. Since `s1` is 5×2 we know that the coefficients of `c1` span $(N-2) = (5-2) = 3$ decomposition levels. Thus, it concatenates the elements needed to populate $3N + 1 = 3(3) + 1 = 10$ approximation and detail coefficient submatrices. Based on `s1`, these submatrices include (a) a 1×1 approximation matrix and three 1×1 detail matrices for decomposition level 3 [see `s1(1, :)` and `s1(2, :)`], (b) three 2×2 detail matrices for level 2 [see `s1(3, :)`], and (c) three 4×4 detail matrices for level 1 [see `s1(4, :)`]. The fifth row of `s1` contains the size of the original image `f`.

Matrix `approx` = 260 is extracted from `c1` using toolbox function `appcoef2`, which has the following syntax:




```
a = appcoef2(c, s, wname)
```

Here, `wname` is a wavelet name from Table 8.1 and `a` is the returned approximation matrix. The horizontal detail coefficients at level 2 are retrieved using `detcoef2`, a function of similar syntax




```
d = detcoef2(o, c, s, n)
```

in which `o` is set to `'h'`, `'v'`, or `'d'` for the horizontal, vertical, and diagonal details and `n` is the desired decomposition level. In this example, 2×2 matrix `horizdet2` is returned. The coefficients corresponding to `horizdet2` in `c1` are then zeroed using `wthcoef2`, a wavelet thresholding function of the form




```
nc = wthcoef2(type, c, s, n, t, sorh)
```

where `type` is set to `'a'` to threshold approximation coefficients and `'h'`, `'v'`, or `'d'` to threshold horizontal, vertical, or diagonal details, respectively. Input `n` is a vector of decomposition levels to be thresholded based on the corresponding thresholds in vector `t`, while `sorh` is set to `'s'` or `'h'` for soft or hard thresholding, respectively. If `t` is omitted, all coefficients meeting the `type` and `n` specifications are zeroed. Output `nc` is the modified (i.e., thresholded) decomposition vector. All three of the preceding Wavelet Toolbox functions have other syntaxes that can be examined using the MATLAB `help` command. ■

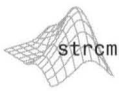
8.3.1 Editing Wavelet Decomposition Coefficients without the Wavelet Toolbox

Without the Wavelet Toolbox, bookkeeping matrix **S** is the key to accessing the individual approximation and detail coefficients of multiscale vector **c**. In this section, we use **S** to build a set of general-purpose routines for the manipulation of **c**. Function `wavework` is the foundation of the routines developed, which are based on the familiar cut-copy-paste metaphor of modern word processing applications.

wavework

```
function [varargout] = wavework(opcode, type, c, s, n, x)
%WAVEWORK is used to edit wavelet decomposition structures.
%   [VARARGOUT] = WAVEWORK(OPCODE, TYPE, C, S, N, X) gets the
%   coefficients specified by TYPE and N for access or modification
%   based on OPCODE.
%
%   INPUTS:
%     OPCODE      Operation to perform
%     ------------------------------------------------------------
%     'copy'      [varargout] = Y = requested (via TYPE and N)
%                 coefficient matrix
%     'cut'       [varargout] = [NC, Y] = New decomposition vector
%                 (with requested coefficient matrix zeroed) AND
%                 requested coefficient matrix
%     'paste'     [varargout] = [NC] = new decomposition vector with
%                 coefficient matrix replaced by X
%
%     TYPE        Coefficient category
%     ----------------------------------------
%     'a'         Approximation coefficients
%     'h'         Horizontal details
%     'v'         Vertical details
%     'd'         Diagonal details
%
%     [C, S] is a wavelet toolbox decomposition structure.
%     N is a decomposition level (Ignored if TYPE = 'a').
%     X is a 2- or 3-D coefficient matrix for pasting.
%
%   See also WAVECUT, WAVECOPY, and WAVEPASTE.

error(nargchk(4, 6, nargin));

if (ndims(c) ~= 2) || (size(c, 1) ~= 1)
   error('C must be a row vector.');
end

if (ndims(s) ~= 2) || ~isreal(s) || ~isnumeric(s) || ...
        ((size(s, 2) ~= 2) && (size(s, 2) ~= 3))
   error('S must be a real, numeric two- or three-column array.');
end
```

```
elements = prod(s, 2);                    % Coefficient matrix elements.
if (length(c) < elements(end)) || ...
      ~(elements(1) + 3 * sum(elements(2:end - 1)) >= elements(end))
    error(['[C S] must form a standard wavelet decomposition ' ...
           'structure.']);
end
```



Function `strcmpi` compares two strings ignoring character case.

```
if strcmpi(opcode(1:3), 'pas') && nargin < 6
   error('Not enough input arguments.');
end

if nargin < 5
   n = 1;                                 % Default level is 1.
end
nmax = size(s, 1) - 2;                    % Maximum levels in [C, S].

aflag = (lower(type(1)) == 'a');
if ~aflag && (n > nmax)
   error('N exceeds the decompositions in [C, S].');
end

switch lower(type(1))                     % Make pointers into C.
case 'a'
   nindex = 1;
   start = 1;    stop = elements(1);    ntst = nmax;
case  {'h', 'v', 'd'}
   switch type
   case 'h', offset = 0;                  % Offset to details.
   case 'v', offset = 1;
   case 'd', offset = 2;
   end
   nindex = size(s, 1) - n;               % Index to detail info.
   start = elements(1) + 3 * sum(elements(2:nmax - n + 1)) + ...
           offset * elements(nindex) + 1;
   stop = start + elements(nindex) - 1;
   ntst = n;
otherwise
   error('TYPE must begin with "a", "h", "v", or "d".');
end

switch lower(opcode)                      % Do requested action.
case {'copy', 'cut'}
   y = c(start:stop);    nc = c;
   y = reshape(y, s(nindex, :));
   if strcmpi(opcode(1:3), 'cut')
      nc(start: stop) = 0; varargout = {nc, y};
   else
      varargout = {y};
   end
case 'paste'
```

```
   if numel(x) ~= elements(end - ntst)
      error('X is not sized for the requested paste.');
   else
      nc = c;   nc(start:stop) = x(:);   varargout = {nc};
   end
otherwise
   error('Unrecognized OPCODE.');
end
```

As wavework checks its input arguments for reasonableness, the number of elements in each coefficient submatrix of c is computed via elements = prod(s, 2). Recall from Section 3.4.2 that MATLAB function Y = prod(X, DIM) computes the products of the elements of X along dimension DIM. The first switch statement then begins the computation of a pair of pointers to the coefficients associated with input parameters type and n. For the approximation case, the computation is trivial since the coefficients are always at the start of c (i.e., start is 1); the ending index is of course the number of elements in the approximation matrix, which is elements(1). When a detail coefficient submatrix is requested, however, start is computed by summing the number of elements at all decomposition levels above n and adding offset * elements(nindex); where offset is 0, 1, or 2 for the horizontal, vertical, or diagonal coefficients, respectively, and nindex is a pointer to the row of s that corresponds to input parameter n.

The second switch statement in function wavework performs the operation requested by opcode. For the 'cut' and 'copy' cases, the coefficients of c between start and stop are copied into vector y, which is then "reshaped" to create a 2-D matrix whose size is determined by s. This is done using y = reshape(y, s(nindex, :)), where the generalized MATLAB function

```
y = reshape(x, m, n)
```



returns an m by n matrix whose elements are taken column wise from x. An error is returned if x does not have m*n elements. For the 'paste' case, the elements of x are copied into c between start and stop. For both the 'cut' and 'paste' operations, a new decomposition vector nc is returned.

The following three functions—wavecut, wavecopy, and wavepaste—use wavework to manipulate c using a more intuitive syntax:

wavecut

```
function [nc, y] = wavecut(type, c, s, n)
%WAVECUT Zeroes coefficients in a wavelet decomposition structure.
%   [NC, Y] = WAVECUT(TYPE, C, S, N) returns a new decomposition
%   vector whose detail or approximation coefficients (based on TYPE
%   and N) have been zeroed. The coefficients that were zeroed are
%   returned in Y.
%
%   INPUTS:
```

```
%     TYPE      Coefficient category
%     ------------------------------------
%     'a'       Approximation coefficients
%     'h'       Horizontal details
%     'v'       Vertical details
%     'd'       Diagonal details
%
%     [C, S] is a wavelet data structure.
%     N specifies a decomposition level (ignored if TYPE = 'a').
%
%   See also WAVEWORK, WAVECOPY, and WAVEPASTE.

error(nargchk(3, 4, nargin));
if nargin == 4
   [nc, y] = wavework('cut', type, c, s, n);
else
   [nc, y] = wavework('cut', type, c, s);
end
```

wavecopy

```
function y = wavecopy(type, c, s, n)
%WAVECOPY Fetches coefficients of a wavelet decomposition structure.
%   Y = WAVECOPY(TYPE, C, S, N) returns a coefficient array based on
%   TYPE and N.
%
%   INPUTS:
%     TYPE      Coefficient category
%     ------------------------------------
%     'a'       Approximation coefficients
%     'h'       Horizontal details
%     'v'       Vertical details
%     'd'       Diagonal details
%
%     [C, S] is a wavelet data structure.
%     N specifies a decomposition level (ignored if TYPE = 'a').
%
%   See also WAVEWORK, WAVECUT, and WAVEPASTE.

error(nargchk(3, 4, nargin));
if nargin == 4
   y = wavework('copy', type, c, s, n);
else
   y = wavework('copy', type, c, s);
end
```

wavepaste

```
function nc = wavepaste(type, c, s, n, x)
%WAVEPASTE Puts coefficients in a wavelet decomposition structure.
%   NC = WAVEPASTE(TYPE, C, S, N, X) returns the new decomposition
%   structure after pasting X into it based on TYPE and N.
%
%   INPUTS:
%     TYPE      Coefficient category
%     ------------------------------------
%     'a'       Approximation coefficients
%     'h'       Horizontal details
%     'v'       Vertical details
%     'd'       Diagonal details
%
%     [C, S] is a wavelet data structure.
%     N specifies a decomposition level (Ignored if TYPE = 'a').
%     X is a 2- or 3-D approximation or detail coefficient
%       matrix whose dimensions are appropriate for decomposition
%       level N.
%
%   See also WAVEWORK, WAVECUT, and WAVECOPY.

error(nargchk(5, 5, nargin))
nc = wavework('paste', type, c, s, n, x);
```

EXAMPLE 8.5: Manipulating **c** with wavecut and wavecopy.

■ Functions wavecopy and wavecut can be used to reproduce the Wavelet Toolbox based results of Example 8.4:

```
>> f = magic(8);
>> [c1, s1] = wavedec2(f, 3, 'haar');
>> approx = wavecopy('a', c1, s1)

approx =
    260.0000

>> horizdet2 = wavecopy('h', c1, s1, 2)
horizdet2 =
    1.0e-013 *
           0   -0.2842
           0         0

>> [newc1, horizdet2] = wavecut('h', c1, s1, 2);
>> newhorizdet2 = wavecopy('h', newc1, s1, 2)
newhorizdet2 =
    0     0
    0     0
```

Note that all extracted matrices are identical to those of the previous example. ■

8.3.2 Displaying Wavelet Decomposition Coefficients

As was indicated in Section 8.3, the coefficients that are packed into one-dimensional wavelet decomposition vector **c** are, in reality, the coefficients of the two-dimensional output arrays from the filter bank in Fig. 8.2. For each iteration of the filter bank, four quarter-size coefficient arrays (neglecting any expansion that may result from the convolution process) are produced. They can be arranged as a 2×2 array of submatrices that replace the two-dimensional input from which they are derived. Function `wavedisplay` performs a similar subimage compositing; it scales the coefficients to better reveal their differences and inserts borders to delineate the approximation and various horizontal, vertical, and diagonal detail matrices.

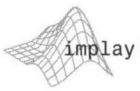
`wavedisplay`

```
function w = wavedisplay(c, s, scale, border)
%WAVEDISPLAY Display wavelet decomposition coefficients.
%   W = WAVEDISPLAY(C, S, SCALE, BORDER) displays and returns a
%   wavelet coefficient image.
%
%   EXAMPLES:
%     wavedisplay(c, s);                      Display w/defaults.
%     foo = wavedisplay(c, s);                Display and return.
%     foo = wavedisplay(c, s, 4);             Magnify the details.
%     foo = wavedisplay(c, s, -4);            Magnify absolute values.
%     foo = wavedisplay(c, s, 1, 'append');   Keep border values.
%
%   INPUTS/OUTPUTS:
%     [C, S] is a wavelet decomposition vector and bookkeeping
%     matrix.
%
%     SCALE       Detail coefficient scaling
%     ------------------------------------------------------
%     0 or 1      Maximum range (default)
%     2,3...      Magnify default by the scale factor
%     -1, -2...   Magnify absolute values by abs(scale)
%
%     BORDER      Border between wavelet decompositions
%     ------------------------------------------------------
%     'absorb'    Border replaces image (default)
%     'append'    Border increases width of image
%
%     Image W:   ------- ------ ------------ -------------------
%               |       |      |            |
%               | a(n)  | h(n) |            |
%               |       |      |            |
%               ------- ------     h(n-1)   |
%               |       |      |            |
%               | v(n)  | d(n) |            |        h(n-2)
%               |       |      |            |
%               ------- ------ ------------
```

```
%                |             |             |
%                |   v(n-1)    |   d(n-1)    |
%                |             |             |
%                -------------- ------------ ------------------
%                |                          |
%                |         v(n-2)           |        d(n-2)
%                |                          |
%
%     Here, n denotes the decomposition step scale and a, h, v, d are
%     approximation, horizontal, vertical, and diagonal detail
%     coefficients, respectively.

% Check input arguments for reasonableness.
error(nargchk(2, 4, nargin));

if (ndims(c) ~= 2) || (size(c, 1) ~= 1)
  error('C must be a row vector.');
end

if (ndims(s) ~= 2) || ~isreal(s) || ~isnumeric(s) || ...
       ((size(s, 2) ~= 2) && (size(s, 2) ~= 3))
  error('S must be a real, numeric two- or three-column array.');
end

elements = prod(s, 2);
if (length(c) < elements(end)) || ...
     ~(elements(1) + 3 * sum(elements(2:end - 1)) >= elements(end))
   error(['[C S] must be a standard wavelet ' ...
          'decomposition structure.']);
end

if (nargin > 2) && (~isreal(scale) || ~isnumeric(scale))
   error('SCALE must be a real, numeric scalar.');
end

if (nargin > 3) && (~ischar(border))
  error('BORDER must be character string.');
end

if nargin == 2
   scale = 1;  % Default scale.
end

if nargin < 4
   border = 'absorb';  % Default border.
end

% Scale coefficients and determine pad fill.
absflag = scale < 0;
scale = abs(scale);
```

```
if scale == 0
   scale = 1;
end

[cd, w] = wavecut('a', c, s);   w = mat2gray(w);
cdx = max(abs(cd(:))) / scale;
if absflag
   cd = mat2gray(abs(cd), [0, cdx]);   fill = 0;
else
   cd = mat2gray(cd, [-cdx, cdx]);   fill = 0.5;
end

% Build gray image one decomposition at a time.
for i = size(s, 1) - 2:-1:1
   ws = size(w);

   h = wavecopy('h', cd, s, i);
   pad = ws - size(h);     frontporch = round(pad / 2);
   h = padarray(h, frontporch, fill, 'pre');
   h = padarray(h, pad - frontporch, fill, 'post');

   v = wavecopy('v', cd, s, i);
   pad = ws - size(v);     frontporch = round(pad / 2);
   v = padarray(v, frontporch, fill, 'pre');
   v = padarray(v, pad - frontporch, fill, 'post');

   d = wavecopy('d', cd, s, i);
   pad = ws - size(d);     frontporch = round(pad / 2);
   d = padarray(d, frontporch, fill, 'pre');
   d = padarray(d, pad - frontporch, fill, 'post');

   % Add 1 pixel white border and concatenate coefficients.
   switch lower(border)
   case 'append'
      w = padarray(w, [1 1], 1, 'post');
      h = padarray(h, [1 0], 1, 'post');
      v = padarray(v, [0 1], 1, 'post');
   case 'absorb'
      w(:, end, :) = 1;   w(end, :, :) = 1;
      h(end, :, :) = 1;   v(:, end, :) = 1;
   otherwise
      error('Unrecognized BORDER parameter.');
   end
   w = [w h; v d];
end

% Display result. If the reconstruction is an extended 2-D array
% with 2 or more pages, display as a time sequence.
if nargout == 0
   if size(s, 2) == 2
```

```
        imshow(w);
    else
        implay(w);
    end
end
```



Function `implay` opens a movie player for showing image sequences.

The "help text" or header section of `wavedisplay` details the structure of generated output image `w`. The subimage in the upper left corner of `w`, for instance, is the approximation array that results from the final decomposition step. It is surrounded—in a clockwise manner—by the horizontal, diagonal, and vertical detail coefficients that were generated during the same decomposition. The resulting array of subimages is then surrounded (again in a clockwise manner) by the detail coefficients of the previous decomposition step; and the pattern continues until all of the scales of decomposition vector `c` are appended to two-dimensional matrix `w`.

The compositing just described takes place within the only `for` loop in `wavedisplay`. After checking the inputs for consistency, `wavecut` is called to remove the approximation coefficients from decomposition vector `c`. These coefficients are then scaled for later display using `mat2gray`. Modified decomposition vector `cd` (i.e., `c` without the approximation coefficients) is then similarly scaled. For positive values of input `scale`, the detail coefficients are scaled so that a coefficient value of 0 appears as middle gray; all necessary padding is performed with a `fill` value of 0.5 (mid-gray). If `scale` is negative, the absolute values of the detail coefficients are displayed with a value of 0 corresponding to black and the pad `fill` value is set to 0. After the approximation and detail coefficients have been scaled for display, the first iteration of the `for` loop extracts the last decomposition step's detail coefficients from `cd` and appends them to `w` (after padding to make the dimensions of the four subimages match and insertion of a one-pixel white border) via the `w = [w h; v d]` statement. This process is then repeated for each scale in `c`. Note the use of `wavecopy` to extract the various detail coefficients needed to form `w`.

EXAMPLE 8.6: Transform coefficient display using `wavedisplay`.

■ The following sequence of commands computes the two-scale DWT of the image in Fig. 8.4 with respect to fourth-order Daubechies' wavelets and displays the resulting coefficients:

```
>> f = imread('vase.tif');
>> [c, s] = wavefast(f, 2, 'db4');
>> wavedisplay(c, s);
>> figure; wavedisplay(c, s, 8);
>> figure; wavedisplay(c, s, -8);
```

The images generated by the final three command lines are shown in Figs. 8.5(a) through (c), respectively. Without additional scaling, the detail coefficient differences in Fig. 8.5(a) are barely visible. In Fig. 8.5(b), the differences are accentuated by multiplying the coefficients by 8. Note the mid-gray padding along the borders of the level 1 coefficient subimages; it was inserted to reconcile

a
b c

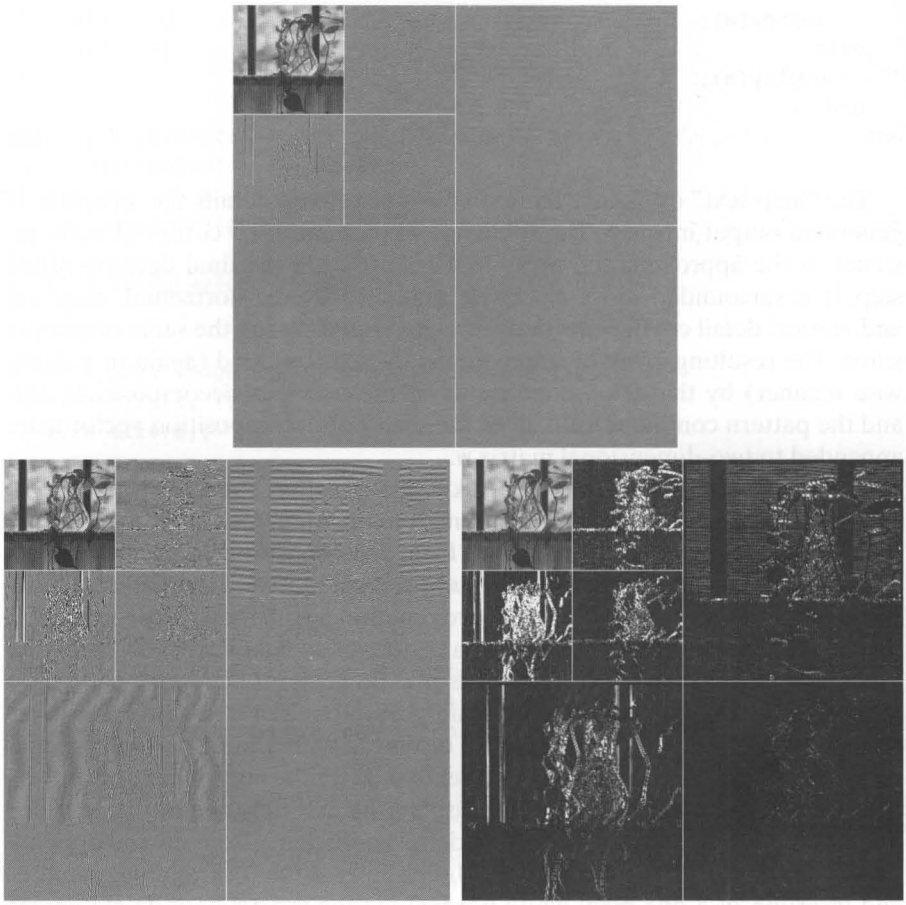


FIGURE 8.5
Displaying a two-scale wavelet transform of the image in Fig. 8.4: (a) Automatic scaling; (b) additional scaling by 8; and (c) absolute values scaled by 8.

dimensional variations between transform coefficient subimages. Figure 8.5(c) shows the effect of taking the absolute values of the details. Here, all padding is done in black. ■

8.4 The Inverse Fast Wavelet Transform

Like its forward counterpart, the *inverse fast wavelet transform* can be computed iteratively using digital filters. Figure 8.6 shows the required *synthesis* or *reconstruction filter bank*, which reverses the process of the analysis or decomposition filter bank of Fig. 8.2. At each iteration, four scale j approximation and detail subimages are *upsampled* (by inserting zeroes between every other element) and convolved with two one-dimension filters—one operating on the subimages' columns and the other on its rows. Addition of the results yields the scale $j+1$ approximation, and the process is repeated until the original image is reconstructed. The filters used in the convolutions are a function of the wavelets employed in the forward transform. Recall that they can be obtained

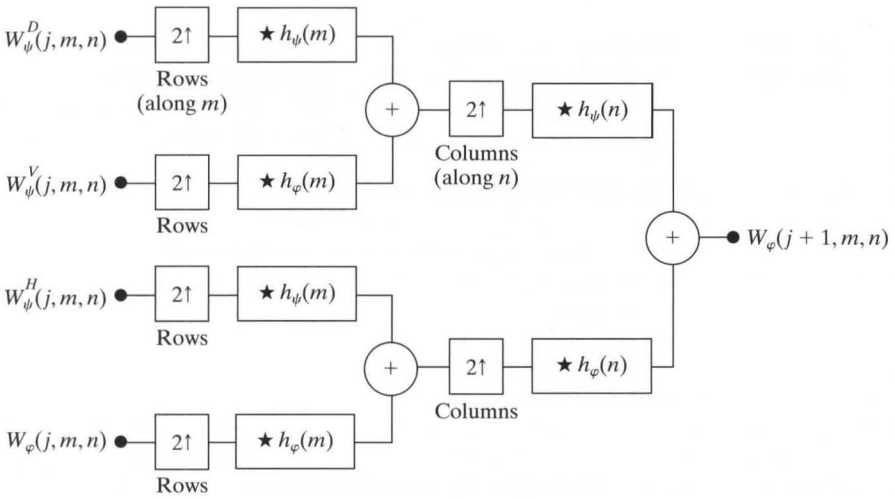


FIGURE 8.6 The 2-D FWT^{-1} filter bank. The boxes with the up arrows represent upsampling by inserting zeroes between every element.

from the `wfilters` and `wavefilter` functions of Section 8.2 with input parameter `type` set to `'r'` for "reconstruction."

When using the Wavelet Toolbox, function `waverec2` is employed to compute the inverse FWT of wavelet decomposition structure `[C, S]`. It is invoked using

```
g = waverec2(C, S, wname)
```



where g is the resulting reconstructed two-dimensional image (of class `double`). The required reconstruction filters can be alternately supplied via syntax

```
g = waverec2(C, S, Lo_R, Hi_R)
```

The following custom routine, which we call `waveback`, can be used when the Wavelet Toolbox is unavailable. It is the final function needed to complete our wavelet-based package for processing images in conjunction with the Image Processing Toolbox (and without the Wavelet Toolbox).

waveback

```
function [varargout] = waveback(c, s, varargin)
%WAVEBACK Computes inverse FWTs for multi-level decomposition [C, S].
%   [VARARGOUT] = WAVEBACK(C, S, VARARGIN) performs a 2D N-level
%   partial or complete wavelet reconstruction of decomposition
%   structure [C, S].
%
%   SYNTAX:
%   Y = WAVEBACK(C, S, 'WNAME');  Output inverse FWT matrix Y
%   Y = WAVEBACK(C, S, LR, HR);   using lowpass and highpass
%                                 reconstruction filters (LR and
%                                 HR) or filters obtained by
```

```
%                                   calling WAVEFILTER with 'WNAME'.
%
%   [NC, NS] = WAVEBACK(C, S, 'WNAME', N);  Output new wavelet
%   [NC, NS] = WAVEBACK(C, S, LR, HR, N);   decomposition structure
%                                           [NC, NS] after N step
%                                           reconstruction.
%
%   See also WAVEFAST and WAVEFILTER.

% Check the input and output arguments for reasonableness.
error(nargchk(3, 5, nargin));
error(nargchk(1, 2, nargout));

if (ndims(c) ~= 2) || (size(c, 1) ~= 1)
   error('C must be a row vector.');
end

if (ndims(s) ~= 2) || ~isreal(s) || ~isnumeric(s) || ...
        ((size(s, 2) ~= 2) && (size(s, 2) ~= 3))
   error('S must be a real, numeric two- or three-column array.');
end

elements = prod(s, 2);
if (length(c) < elements(end)) || ...
      ~(elements(1) + 3 * sum(elements(2:end - 1)) >= elements(end))
   error(['[C S] must be a standard wavelet ' ...
         'decomposition structure.']);
end

% Maximum levels in [C, S].
nmax = size(s, 1) - 2;

% Get third input parameter and init check flags.
wname = varargin{1};  filterchk = 0;   nchk = 0;

switch nargin
case 3
   if ischar(wname)
      [lp, hp] = wavefilter(wname, 'r');   n = nmax;
   else
      error('Undefined filter.');
   end
   if nargout ~= 1
      error('Wrong number of output arguments.');
   end
case 4
   if ischar(wname)
      [lp, hp] = wavefilter(wname, 'r');
      n = varargin{2};    nchk = 1;
   else
```

```
      lp = varargin{1};   hp = varargin{2};
      filterchk = 1;      n = nmax;
      if nargout ~= 1
         error('Wrong number of output arguments.');
      end
   end
case 5
   lp = varargin{1};   hp = varargin{2};   filterchk = 1;
   n = varargin{3};    nchk = 1;
otherwise
   error('Improper number of input arguments.');
end

fl = length(lp);
if filterchk                                    % Check filters.
   if (ndims(lp) ~= 2) || ~isreal(lp) || ~isnumeric(lp) ...
         || (ndims(hp) ~= 2) || ~isreal(hp) || ~isnumeric(hp) ...
         || (fl ~= length(hp)) || rem(fl, 2) ~= 0
      error(['LP and HP must be even and equal length real, ' ...
         'numeric filter vectors.']);
   end
end

if nchk && (~isnumeric(n) || ~isreal(n))        % Check scale N.
   error('N must be a real numeric.');
end
if (n > nmax) || (n < 1)
   error('Invalid number (N) of reconstructions requested.');
end
if (n ~= nmax) && (nargout ~= 2)
   error('Not enough output arguments.');
end

nc = c;    ns = s;    nnmax = nmax;             % Init decomposition.
for i = 1:n
   % Compute a new approximation.
   a = symconvup(wavecopy('a', nc, ns), lp, lp, fl, ns(3, :)) + ...
       symconvup(wavecopy('h', nc, ns, nnmax), ...
                 hp, lp, fl, ns(3, :)) + ...
       symconvup(wavecopy('v', nc, ns, nnmax), ...
                 lp, hp, fl, ns(3, :)) + ...
       symconvup(wavecopy('d', nc, ns, nnmax), ...
                 hp, hp, fl, ns(3, :));

   % Update decomposition.
   nc = nc(4 * prod(ns(1, :)) + 1:end);     nc = [a(:)' nc];
   ns = ns(3:end, :);                       ns = [ns(1, :); ns];
   nnmax = size(ns, 1) - 2;
end
```

```
% For complete reconstructions, reformat output as 2-D.
if nargout == 1
   a = nc;   nc = repmat(0, ns(1, :));     nc(:) = a;
end

varargout{1} = nc;
if nargout == 2
   varargout{2} = ns;
end

%-----------------------------------------------------------------------%
function w = symconvup(x, f1, f2, fln, keep)
% Upsample rows and convolve columns with f1; upsample columns and
% convolve rows with f2; then extract center assuming symmetrical
% extension.

% Process each "page" (i.e., 3rd index) of an extended 2-D array
% separately; if 'x' is 2-D, size(x, 3) = 1.
% Preallocate w.
zi = fln - 1:fln + keep(1) - 2;
zj = fln - 1:fln + keep(2) - 2;
w = zeros(numel(zi), numel(zj), size(x, 3));
for i = 1:size(x, 3)
   y = zeros([2 1] .* size(x(:, :, i)));
   y(1:2:end, :) = x(:, :, i);
   y = conv2(y, f1');
   z = zeros([1 2] .* size(y));      z(:, 1:2:end) = y;
   z = conv2(z, f2);
   z = z(zi, zj);
   w(:, :, i) = z;
end
```

The main routine of function `waveback` is a simple `for` loop that iterates through the requested number of decomposition levels (i.e., scales) in the desired reconstruction. As can be seen, each loop calls internal function `symconvup` four times and sums the returned matrices. Decomposition vector `nc`, which is initially set to `c`, is iteratively updated by replacing the four coefficient matrices passed to `symconvup` by the newly created approximation `a`. Bookkeeping matrix `ns` is then modified accordingly—there is now one less scale in decomposition structure `[nc, ns]`. This sequence of operations is slightly different than the ones outlined in Fig. 8.6, in which the top two inputs are combined to yield

$$\left[W_\psi^D(j,m,n)\uparrow^{2m} \star h_\psi(m) + W_\psi^V(j,m,n)\uparrow^{2m} \star h_\varphi(m)\right]\uparrow^{2n} \star h_\psi(n)$$

where $\uparrow^{2m}$ and $\uparrow^{2n}$ denote upsampling along m and n, respectively. Function `waveback` uses the equivalent computation

$$\left[W_\psi^D(j,m,n)\uparrow^{2m} \star h_\psi(m)\right]\uparrow^{2n} \star h_\psi(n) + \left[W_\psi^V(j,m,n)\uparrow^{2m} \star h_\varphi(m)\right]\uparrow^{2n} \star h_\psi(n)$$

Function symconvup performs the convolutions and upsampling required to compute the contribution of one input of Fig. 8.6 to output $W_\varphi(j+1, m, n)$ in accordance with the preceding equation. Input x is first upsampled in the row direction to yield y, which is convolved columnwise with filter f1. The resulting output, which replaces y, is then upsampled in the column direction and convolved row by row with f2 to produce z. Finally, the center keep elements of z (the final convolution) are returned as input x's contribution to the new approximation.

EXAMPLE 8.7: Comparing the execution times of waveback and waverec2.

■ The following test routine compares the execution times of Wavelet Toolbox function waverec2 and custom function waveback using a simple modification of the test function in Example 8.3:

```
function [ratio, maxdiff] = ifwtcompare(f, n, wname)
%IFWTCOMPARE Compare waverec2 and waveback.
%   [RATIO, MAXDIFF] = IFWTCOMPARE(F, N, WNAME) compares the
%   operation of Wavelet Toolbox function WAVEREC2 and custom
%   function WAVEBACK.
%
%   INPUTS:
%     F           Image to transform and inverse transform.
%     N           Number of scales to compute.
%     WNAME       Wavelet to use.
%
%   OUTPUTS:
%     RATIO       Execution time ratio (custom/toolbox).
%     MAXDIFF     Maximum generated image difference.

% Compute the transform and get output and computation time for
% waverec2.
[c1, s1] = wavedec2(f, n, wname);
w1 = @() waverec2(c1, s1, wname);
reftime = timeit(w1);

% Compute the transform and get output and computation time for
% waveback.
[c2, s2] = wavefast(f, n, wname);
w2 = @() waveback(c2, s2, wname);
t2 = timeit(w2);

% Compare the results.
ratio = t2 / reftime;
diff = double(w1()) - w2();
maxdiff = abs(max(diff(:)));
```

For a five scale transform of the 512×512 image in Fig. 8.4 with respect to 4th order Daubechies' wavelets, we get

```
>> f = imread('vase.tif');
>> [ratio, maxdifference] = ifwtcompare(f, 5, 'db4')
ratio =
    1.2238
maxdifference =
    3.6948e-013
```

Note that the inverse transformation times of the two functions are similar (i.e., the ratio is 1.2238) and that the largest output difference is 3.6948×10^{-13}. For all practical purposes, they essentially equivalent. ■

8.5 Wavelets in Image Processing

As in the Fourier domain (see Section 4.3.2), the basic approach to wavelet-based image processing is to

1. Compute the two-dimensional wavelet transform of an image.
2. Alter the transform coefficients.
3. Compute the inverse transform.

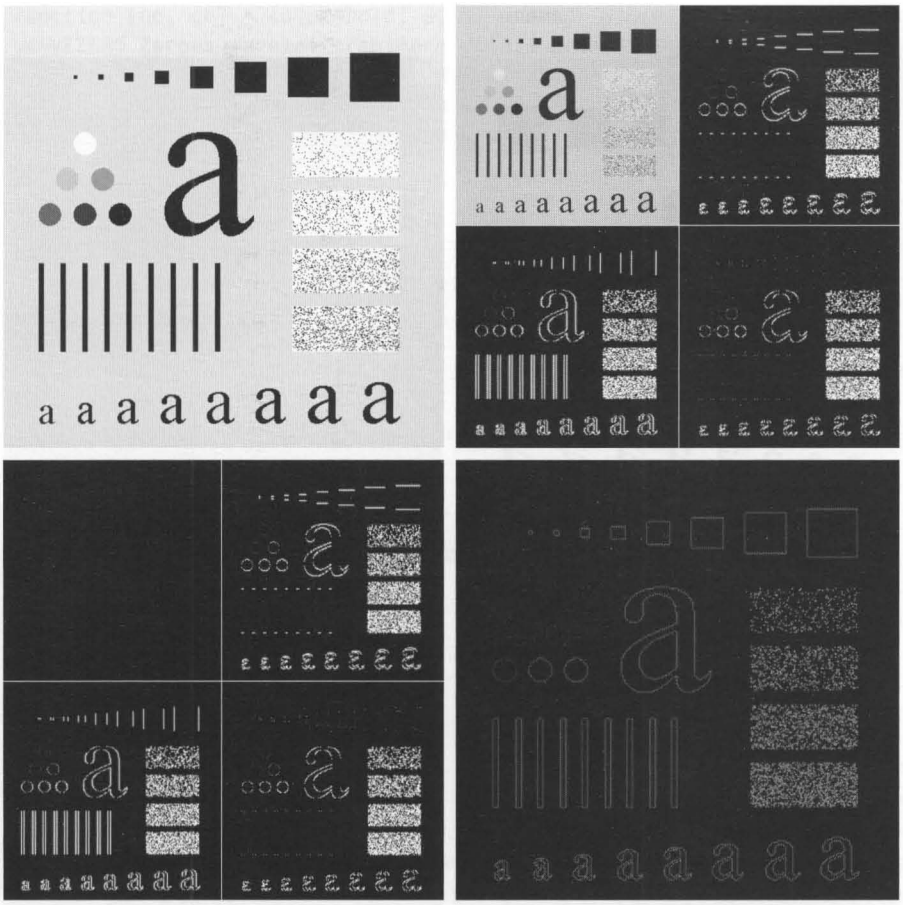
Because scale in the wavelet domain is analogous to frequency in the Fourier domain, most of the Fourier-based filtering techniques of Chapter 4 have an equivalent "wavelet domain" counterpart. In this section, we use the preceding three-step procedure to give several examples of the use of wavelets in image processing. Attention is restricted to the routines developed earlier in the chapter; the Wavelet Toolbox is not needed to implement the examples given here—nor the examples in Chapter 7 of *Digital Image Processing* (Gonzalez and Woods [2008]).

EXAMPLE 8.8: Wavelet directionality and edge detection.

■ Consider the 500×500 test image in Fig. 8.7(a). This image was used in Chapter 4 to illustrate smoothing and sharpening with Fourier transforms. Here, we use it to demonstrate the directional sensitivity of the 2-D wavelet transform and its usefulness in edge detection:

```
>> f = imread('A.tif');
>> imshow(f);
>> [c, s] = wavefast(f, 1, 'sym4');
>> figure; wavedisplay(c, s, -6);
>> [nc, y] = wavecut('a', c, s);
>> figure; wavedisplay(nc, s, -6);
>> edges = abs(waveback(nc, s, 'sym4'));
>> figure; imshow(mat2gray(edges));
```

The horizontal, vertical, and diagonal directionality of the single-scale wavelet transform of Fig. 8.7(a) with respect to `'sym4'` wavelets is clearly visible in Fig. 8.7(b). Note, for example, that the horizontal edges of the original image are present in the horizontal detail coefficients of the upper-right quad-



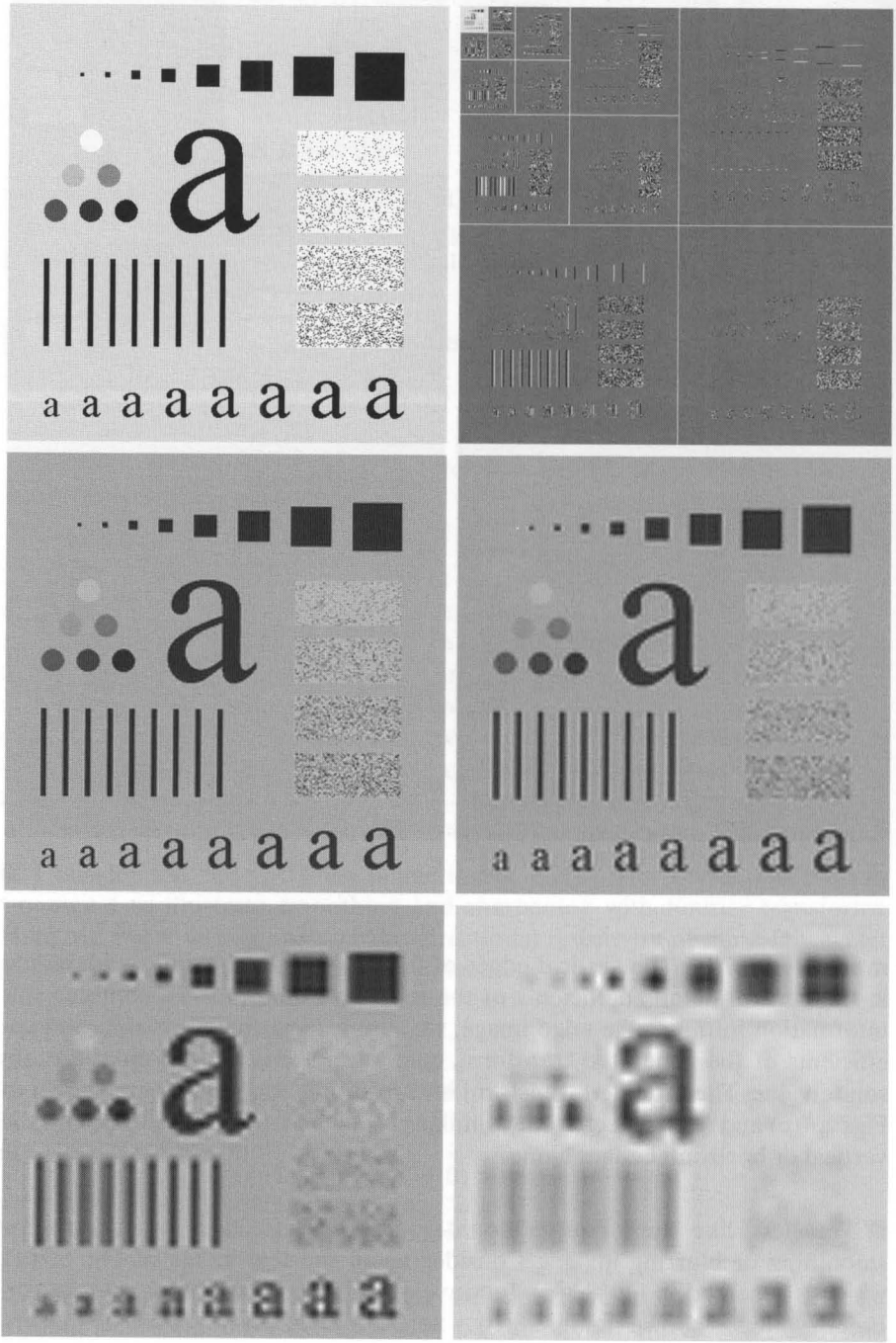
a b
c d

FIGURE 8.7
Wavelets in edge detection: (a) A simple test image; (b) its wavelet transform; (c) the transform modified by zeroing all approximation coefficients; and (d) the edge image resulting from computing the absolute value of the inverse transform.

rant of Fig. 8.7(b). The vertical edges of the image can be similarly identified in the vertical detail coefficients of the lower-left quadrant. To combine this information into a single edge image, we simply zero the approximation coefficients of the generated transform, compute its inverse, and take the absolute value. The modified transform and resulting edge image are shown in Figs. 8.7(c) and (d), respectively. A similar procedure can be used to isolate the vertical or horizontal edges alone. ■

■ Wavelets, like their Fourier counterparts, are effective instruments for smoothing or blurring images. Consider again the test image of Fig. 8.7(a), which is repeated in Fig. 8.8(a). Its wavelet transform with respect to fourth-order symlets is shown in Fig. 8.8(b), where it is clear that a four-scale decomposition has been performed. To streamline the smoothing process, we employ the following utility function:

EXAMPLE 8.9: Wavelet-based image smoothing or blurring.



a b
c d
e f

FIGURE 8.8
Wavelet-based image smoothing: (a) A test image; (b) its wavelet transform; (c) the inverse transform after zeroing the first level detail coefficients; and (d) through (f) similar results after zeroing the second-, third-, and fourth-level details.

wavezero

```
function [nc, g8] = wavezero(c, s, l, wname)
%WAVEZERO Zeroes wavelet transform detail coefficients.
%   [NC, G8] = WAVEZERO(C, S, L, WNAME) zeroes the level L detail
%   coefficients in wavelet decomposition structure [C, S] and
%   computes the resulting inverse transform with respect to WNAME
%   wavelets.

[nc, foo] = wavecut('h', c, s, l);
[nc, foo] = wavecut('v', nc, s, l);
[nc, foo] = wavecut('d', nc, s, l);
i = waveback(nc, s, wname);
g8 = im2uint8(mat2gray(i));
figure; imshow(g8);
```

Using `wavezero`, a series of increasingly smoothed versions of Fig. 8.8(a) can be generated with the following sequence of commands:

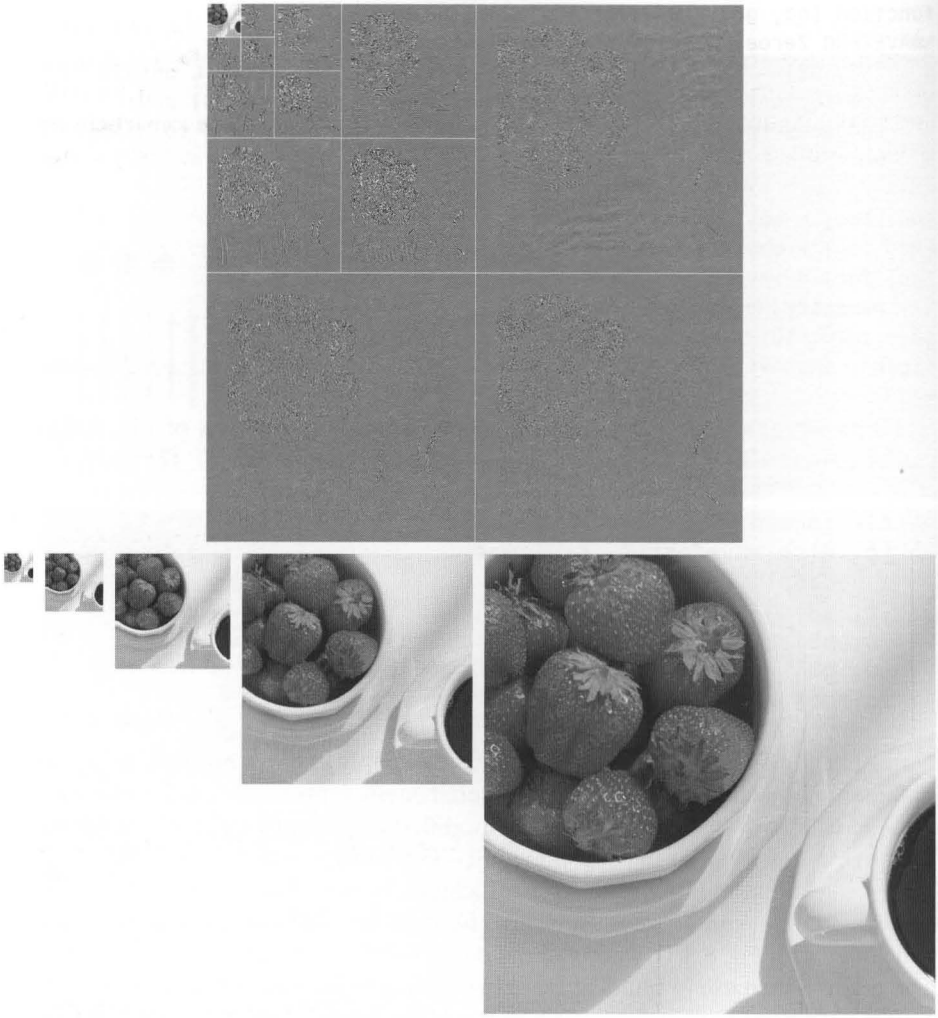
```
>> f = imread('A.tif');
>> [c, s] = wavefast(f, 4, 'sym4');
>> wavedisplay(c, s, 20);
>> [c, g8] = wavezero(c, s, 1, 'sym4');
>> [c, g8] = wavezero(c, s, 2, 'sym4');
>> [c, g8] = wavezero(c, s, 3, 'sym4');
>> [c, g8] = wavezero(c, s, 4, 'sym4');
```

Note that the smoothed image in Fig. 8.8(c) is only slightly blurred, as it was obtained by zeroing only the first-level detail coefficients of the original image's wavelet transform (and computing the modified transform's inverse). Additional blurring is present in the second result—Fig. 8.8(d)—which shows the effect of zeroing the second level detail coefficients as well. The coefficient zeroing process continues in Fig. 8.8(e), where the third level of details is zeroed, and concludes with Fig. 8.8(f), where all the detail coefficients have been eliminated. The gradual increase in blurring from Figs. 8.8(c) to (f) is reminiscent of similar results with Fourier transforms. It illustrates the intimate relationship between scale in the wavelet domain and frequency in the Fourier domain. ■

EXAMPLE 8.10: Progressive reconstruction.

■ Consider next the transmission and reconstruction of the four-scale wavelet transform in Fig. 8.9(a) within the context of browsing a remote image database for a specific image. Here, we deviate from the three-step procedure described at the beginning of this section and consider an application without a Fourier domain counterpart. Each image in the database is stored as a multiscale wavelet decomposition. This structure is well suited to progressive reconstruction applications, particularly when the 1-D decomposition vector used to store the transform's coefficients assumes the general format of Section 8.3. For the four-scale transform of this example, the decomposition vector is

$$[\mathbf{A}_4(:)' \quad \mathbf{H}_4(:)' \quad \cdots \quad \mathbf{H}_i(:)' \quad \mathbf{V}_i(:)' \quad \mathbf{D}_i(:)' \quad \ldots \quad \mathbf{V}_1(:)' \quad \mathbf{D}_1(:)']$$



a
b c d e f

FIGURE 8.9 Progressive reconstruction: (a) A four-scale wavelet transform; (b) the fourth-level approximation image from the upper-left corner; (c) a refined approximation incorporating the fourth-level details; (d) through (f) further resolution improvements incorporating higher-level details.

where $\mathbf{A}_4$ is the approximation coefficient matrix of the fourth decomposition level and $\mathbf{H}_i$, $\mathbf{V}_i$, and $\mathbf{D}_i$ for $i = 1, 2, 3, 4$ are the horizontal, vertical, and diagonal transform coefficient matrices for level i. If we transmit this vector in a left-to-right manner, a remote display device can gradually build higher resolution approximations of the final high-resolution image (based on the user's needs) as the data arrives at the viewing station. For instance, when the

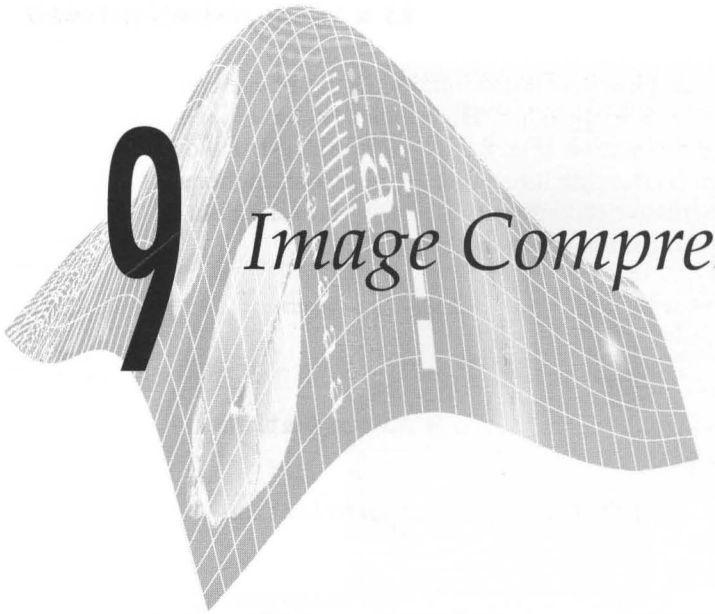
$\mathbf{A}_4$ coefficients have been received, a low-resolution version of the image can be made available for viewing [Fig. 8.9(b)]. When $\mathbf{H}_4$, $\mathbf{V}_4$, and $\mathbf{D}_4$ have been received, a higher-resolution approximation [Fig. 8.9(c)] can be constructed, and so on. Figures 8.9(d) through (f) provide three additional reconstructions of increasing resolution. This progressive reconstruction process is easily simulated using the following MATLAB command sequence:

```
>> f = imread('Strawberries.tif');          % Transform
>> [c, s] = wavefast(f, 4, 'jpeg9.7');
>> wavedisplay(c, s, 8);
>>
>> f = wavecopy('a', c, s);                 % Approximation 1
>> figure; imshow(mat2gray(f));
>>
>> [c, s] = waveback(c, s, 'jpeg9.7', 1);   % Approximation 2
>> f = wavecopy('a', c, s);
>> figure; imshow(mat2gray(f));
>> [c, s] = waveback(c, s, 'jpeg9.7', 1);   % Approximation 3
>> f = wavecopy('a', c, s);
>> figure; imshow(mat2gray(f));
>> [c, s] = waveback(c, s, 'jpeg9.7', 1);   % Approximation 4
>> f = wavecopy('a', c, s);
>> figure; imshow(mat2gray(f));
>> [c, s] = waveback(c, s, 'jpeg9.7', 1);   % Final image
>> f = wavecopy('a', c, s);
>> figure; imshow(mat2gray(f));
```

Note that the final four approximations use `waveback` to perform single level reconstructions. ■

Summary

The material in this chapter introduces the wavelet transform and its use in image processing. Like the Fourier transform, wavelet transforms can be used in tasks ranging from edge detection to image smoothing, both of which are considered in the material that is covered. Because they provide significant insight into both an image's spatial and frequency characteristics, wavelets can also be used in applications in which Fourier methods are not well suited, like progressive image reconstruction (see Example 8.10). Because the Image Processing Toolbox does not include routines for computing or using wavelet transforms, a significant portion of this chapter is devoted to the development of a set of functions that extend the Image Processing Toolkit to wavelet-based imaging. The functions developed were designed to be fully compatible with MATLAB's Wavelet Toolbox, which is introduced in this chapter but is not a part of the Image Processing Toolbox. In the next chapter, wavelets will be used for image compression, an area in which they have received considerable attention in the literature.



9 Image Compression

Preview

Image compression addresses the problem of reducing the amount of data required to represent a digital image. Compression is achieved by the removal of one or three basic data *redundancies*: (1) *coding redundancy*, which is present when less than optimal (i.e., the smallest length) code words are used; (2) *spatial and/or temporal redundancy*, which results from correlations between the pixels of an image or between the pixels of neighboring images in a sequence of images; and (3) *irrelevant information*, which is due to data that is ignored by the human visual system (i.e., visually nonessential information). In this chapter, we examine each of these redundancies, describe a few of the many techniques that can be used to exploit them, and examine two important compression standards—JPEG and JPEG 2000. These standards unify the concepts introduced earlier in the chapter by combining techniques that collectively attack all three data redundancies.

Because the Image Processing Toolbox does not include functions for image compression, a major goal of this chapter is to provide practical ways of exploring compression techniques within the context of MATLAB. For instance, we develop a MATLAB callable C function that illustrates how to manipulate variable-length data representations at the bit level. This is important because variable-length coding is a mainstay of image compression, but MATLAB is best at processing matrices of uniform (i.e., fixed length) data. During the development of the function, we assume that the reader has a working knowledge of the C language and focus our discussion on how to make MATLAB interact with programs (both C and Fortran) external to the MATLAB environment. This is an important skill when there is a need to interface M-functions to preexisting C or Fortran programs, and when vectorized M-functions still need to be speeded up (e.g., when a `for` loop can not be adequately vectorized). In the end, the range of compression functions developed in this chapter, together

with MATLAB's ability to treat C and Fortran programs as though they were conventional M-files or built-in functions, demonstrates that MATLAB can be an effective tool for prototyping image compression systems and algorithms.

9.1 Background

As can be seen in Fig. 9.1, image compression systems are composed of two distinct structural blocks: an *encoder* and a *decoder*. Image $f(x, y)$ is fed into the encoder, which creates a set of symbols from the input data and uses them to represent the image. If we let n_1 and n_2 denote the number of information carrying units (usually bits) in the original and encoded images, respectively, the compression that is achieved can be quantified numerically via the *compression ratio*

In video compression systems, $f(x, y)$ would be replaced by $f(x, y, t)$ and frames would be sequentially fed into the block diagram of Fig. 9.1.

$$C_R = \frac{n_1}{n_2}$$

A compression ratio like 10 (or 10:1) indicates that the original image has 10 information carrying units (e.g., bits) for every 1 unit in the compressed data set. In MATLAB, the ratio of the number of bits used in the representation of two image files and/or variables can be computed with the following M-function:

imratio

```
function cr = imratio(f1, f2)
%IMRATIO Computes the ratio of the bytes in two images/variables.
%   CR = IMRATIO(F1, F2) returns the ratio of the number of bytes in
%   variables/files F1 and F2. If F1 and F2 are an original and
%   compressed image, respectively, CR is the compression ratio.

error(nargchk(2, 2, nargin));          % Check input arguments
cr = bytes(f1) / bytes(f2);            % Compute the ratio

%--------------------------------------------------------------------%
function b = bytes(f)
% Return the number of bytes in input f. If f is a string, assume
```

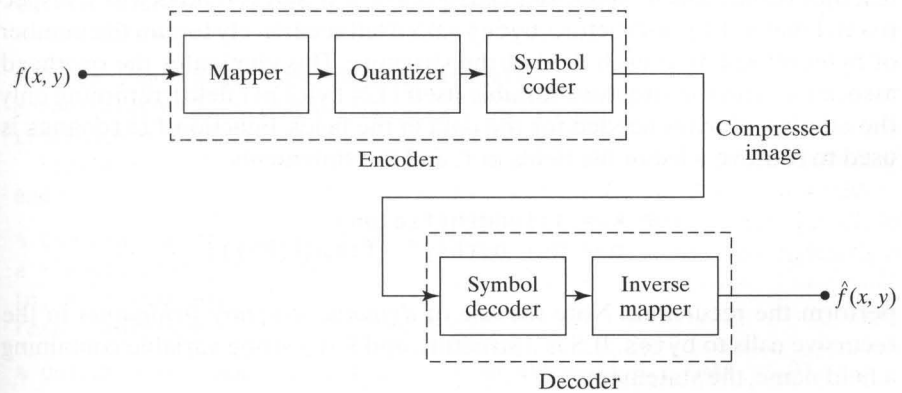


FIGURE 9.1 A general image compression system block diagram.

```
% that it is an image filename; if not, it is an image variable.

if ischar(f)
   info = dir(f);        b = info.bytes;
elseif isstruct(f)
   % MATLAB's whos function reports an extra 124 bytes of memory
   % per structure field because of the way MATLAB stores
   % structures in memory.  Don't count this extra memory; instead,
   % add up the memory associated with each field.
   b = 0;
   fields = fieldnames(f);
   for k = 1:length(fields)
      elements = f.(fields{k});
      for m = 1:length(elements)
          b = b + bytes(elements(m));
      end
   end
else
   info = whos('f');     b = info.bytes;
end
```

For example, the compression of the JPEG encoded image in Fig. 2.5(c) of Chapter 2 can be computed via

```
>> r = imratio(imread('bubbles25.jpg'), 'bubbles25.jpg')
r =
    35.1612
```

Note that in function `imratio`, internal function `b = bytes(f)` is designed to return the number of bytes in (1) a file, (2) a structure variable, and/or (3) a nonstructure variable. If `f` is a nonstructure variable, function `whos`, introduced in Section 2.2, is used to get its size in bytes. If `f` is a file name, function `dir` performs a similar service. In the syntax employed, `dir` returns a structure (see Section 2.10.6 for more on structures) with fields `name`, `date`, `bytes`, and `isdir`. They contain the file's name, modification date, size in bytes, and whether or not it is a directory (`isdir` is 1 if it is and is 0 otherwise), respectively. Finally, if `f` is a structure, `bytes` calls itself recursively to sum the number of bytes allocated to each field of the structure. This eliminates the overhead associated with the structure variable itself (124 bytes per field), returning only the number of bytes needed for the data in the fields. Function `fieldnames` is used to retrieve a list of the fields in `f`, and the statements

```
for k = 1:length(fields)
    b = b + bytes(f.(fields{k}));
```

perform the recursions. Note the use of *dynamic structure fieldnames* in the recursive calls to `bytes`. If S is a structure and `F` is a string variable containing a field name, the statements

```
S.(F) = foo;
field = S.(F);
```

employ the dynamic structure fieldname syntax to set and/or get the contents of structure field `F`, respectively.

To view and/or use a compressed (i.e., encoded) image, it must be fed into a decoder (see Fig. 9.1), where a reconstructed output image, $\hat{f}(x, y)$ is generated. In general, $\hat{f}(x, y)$ may or may not be an exact representation of $f(x, y)$. If it is, the system is called *error free*, *information preserving*, or *lossless*; if not, some level of distortion is present in the reconstructed image. In the latter case, which is called *lossy compression*, we can define the error $e(x, y)$ between $f(x, y)$ and $\hat{f}(x, y)$ for any value of x and y as

In video compression systems, these equations are used to compute the error for a single frame.

$$e(x, y) = \hat{f}(x, y) - f(x, y)$$

so that the total error between the two images is

$$\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [\hat{f}(x, y) - f(x, y)]$$

and the *rms* (*root mean square*) *error* e_{rms} between $f(x, y)$ and $\hat{f}(x, y)$ is the square root of the squared error averaged over the $M \times N$ array, or

$$e_{\text{rms}} = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [\hat{f}(x, y) - f(x, y)]^2 \right]^{1/2}$$

The following M-function computes e_{rms} and displays (if $e_{\text{rms}} \neq 0$) both $e(x, y)$ and its histogram. Since $e(x, y)$ can contain both positive and negative values, `hist` rather than `imhist` (which handles only image data) is used to generate the histogram.

compare

```
function rmse = compare(f1, f2, scale)
%COMPARE Computes and displays the error between two matrices.
%   RMSE = COMPARE(F1, F2, SCALE) returns the root-mean-square error
%   between inputs F1 and F2, displays a histogram of the difference,
%   and displays a scaled difference image. When SCALE is omitted, a
%   scale factor of 1 is used.

% Check input arguments and set defaults.
error(nargchk(2, 3, nargin));
if nargin < 3
   scale = 1;
end

% Compute the root-mean-square error.
e = double(f1) - double(f2);
[m, n] = size(e);
rmse = sqrt(sum(e(:) .^ 2) / (m * n));

% Output error image & histogram if an error (i.e., rmse ~= 0).
```

```
if rmse
   % Form error histogram.
   emax = max(abs(e(:)));
   [h, x] = hist(e(:), emax);
   if length(h) >= 1
      figure; bar(x, h, 'k');

      % Scale the error image symmetrically and display
      emax = emax / scale;
      e = mat2gray(e, [-emax, emax]);
      figure; imshow(e);
   end
end
```

Finally, we note that the encoder of Fig. 9.1 is responsible for reducing the coding, interpixel, and/or psychovisual redundancies of the input image. In the first stage of the encoding process, the *mapper* transforms the input image into a (usually nonvisual) format designed to reduce interpixel redundancies. The second stage, or *quantizer* block, reduces the accuracy of the mapper's output in accordance with a predefined fidelity criterion—attempting to eliminate only psychovisually redundant data. This operation is irreversible and must be omitted when error-free compression is desired. In the third and final stage of the process, a *symbol coder* creates a code (that reduces coding redundancy) for the quantizer output and maps the output in accordance with the code.

The decoder in Fig. 9.1 contains only two components: a *symbol decoder* and an *inverse mapper*. These blocks perform, in reverse order, the inverse operations of the encoder's symbol coder and mapper blocks. Because quantization is irreversible, an inverse quantization block is not included.

9.2 Coding Redundancy

Let the discrete random variable r_k for $k = 1, 2, \ldots, L$ with associated probabilities $p_r(r_k)$ represent the gray levels of an L-gray-level image. As in Chapter 3, r_1 corresponds to gray level 0 (since MATLAB array indices cannot be 0) and

$$p_r(r_k) = \frac{n_k}{n} \quad k = 1, 2, \ldots, L$$

where n_k is the number of times that the kth gray level appears in the image and n is the total number of pixels in the image. If the number of bits used to represent each value of r_k is $l(r_k)$, then the average number of bits required to represent each pixel is

$$L_{\text{avg}} = \sum_{k=1}^{L} l(r_k) p_r(r_k)$$

That is, the average length of the code words assigned to the various gray-level values is found by summing the product of the number of bits used to repre-

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
r_1	0.1875	00	2	011	3
r_2	0.5000	01	2	1	1
r_3	0.1250	10	2	010	3
r_4	0.1875	11	2	00	2

TABLE 9.1 Illustration of coding redundancy: $L_{avg} = 2$ for Code 1; $L_{avg} = 1.81$ for Code 2.

sent each gray level and the probability that the gray level occurs. Thus the total number of bits required to code an $M \times N$ image is MNL_{avg}.

When the gray levels of an image are represented using a natural m-bit binary code, the right-hand side of the preceding equation reduces to m bits. That is, $L_{avg} = m$ when m is substituted for $l(r_k)$. Then the constant m may be taken outside the summation, leaving only the sum of the $p_r(r_k)$ for $1 \leq k \leq L$, which, of course, equals 1. As is illustrated in Table 9.1, coding redundancy is almost always present when the gray levels of an image are coded using a natural binary code. In the table, both a fixed and variable-length encoding of a four-level image whose gray-level distribution is shown in column 2 is given. The 2-bit binary encoding (Code 1) in column 3 has an average length of 2 bits. The average number of bits required by Code 2 (in column 5) is

$$\begin{aligned} L_{avg} &= \sum_{k=1}^{4} l_2(k) p_r(r_k) \\ &= 3(0.1875) + 1(0.5) + 3(0.125) + 2(0.1875) = 1.8125 \end{aligned}$$

and the resulting compression ratio is $C_r = 2/1.8125 \approx 1.103$. The underlying basis for the compression achieved by Code 2 is that its code words are of varying length, allowing the shortest code words to be assigned to the gray levels that occur most frequently in the image.

The question that naturally arises is: How few bits actually are needed to represent the gray levels of an image? That is, is there a minimum amount of data that is sufficient to describe completely an image without loss of information? *Information theory* provides the mathematical framework to answer this and related questions. Its fundamental premise is that the generation of information can be modeled as a probabilistic process that can be measured in a manner that agrees with intuition. In accordance with this supposition, a random event E with probability $P(E)$ is said to contain

$$I(E) = \log \frac{1}{P(E)} = -\log P(E)$$

units of information. If $P(E) = 1$ (that is, the event always occurs), $I(E) = 0$ and no information is attributed to it. That is, because no uncertainty is associated with the event, no information would be transferred by communicating that the event has occurred. Given a source of random events from the discrete set of possible events $\{a_1, a_2, \ldots, a_J\}$ with associated probabilities

$\{P(a_1), P(a_2), \ldots, P(a_J)\}$ the average information per source output, called the *entropy* of the source, is

$$H = -\sum_{j=1}^{J} P(a_j) \log P(a_j)$$

If an image is interpreted as a sample of a "gray-level source" that emitted it, we can model that source's symbol probabilities using the gray-level histogram of the observed image and generate an estimate, called the *first-order estimate*, $\tilde{H}$ of the source's entropy:

$$\tilde{H} = -\sum_{k=1}^{L} p_r(r_k) \log p_r(r_k)$$

Such an estimate is computed by the following M-function and, under the assumption that each gray level is coded independently, is a lower bound on the compression that can be achieved through the removal of coding redundancy alone.

`ntrop`

Note that `ntrop` is similar but not identical to toolbox function `e = entropy(i)`, which computes the entropy of `i` after converting it to `uint8` (with 256 gray levels and 256 histogram bins).

```
function h = ntrop(x, n)
%NTROP Computes a first-order estimate of the entropy of a matrix.
%   H = NTROP(X, N) returns the entropy of matrix X with N
%   symbols. N = 256 if omitted but it must be larger than the
%   number of unique values in X for accurate results. The estimate
%   assumes a statistically independent source characterized by the
%   relative frequency of occurrence of the elements in X.
%   The estimate is a lower bound on the average number of bits per
%   unique value (or symbol) when coding without coding redundancy.
error(nargchk(1, 2, nargin));          % Check input arguments
if nargin < 2
   n = 256;                            % Default for n.
end

x = double(x);                         % Make input double
xh = hist(x(:), n);                    % Compute N-bin histogram
xh = xh / sum(xh(:));                  % Compute probabilities

% Make mask to eliminate 0's since log2(0) = -inf.
i = find(xh);

h = -sum(xh(i) .* log2(xh(i)));        % Compute entropy
```

Note the use of the MATLAB `find` function, which is employed to determine the indices of the nonzero elements of histogram `xh`. The statement `find(x)` is equivalent to `find(x ~= 0)`. Function `ntrop` uses `find` to create a vector of indices, `i`, into histogram `xh`, which is subsequently employed to eliminate all zero-valued elements from the entropy computation in the final statement. If this were not done, the `log2` function would force output `h` to `NaN(0 * -inf` is *not a number*) when any symbol probability was 0.

EXAMPLE 9.1: Computing entropy.

■ Consider a simple 4×4 image whose histogram (see `p` in the following code) models the symbol probabilities in Table 9.1. The following command line sequence generates one such image and computes a first-order estimate of its entropy.

```
>> f = [119 123 168 119; 123 119 168 168];
>> f = [f; 119 119 107 119; 107 107 119 119]
f =
    119   123   168   119
    123   119   168   168
    119   119   107   119
    107   107   119   119
p = hist(f(:), 8);
p = p / sum(p)
p =
    0.1875   0.5   0.125   0   0   0   0   0.1875
h = ntrop(f)
h =
    1.7806
```

Code 2 of Table 9.1, with $L_{avg} \approx 1.81$, approaches this first-order entropy estimate and is a minimal length *binary* code for image `f`. Note that gray level 107 corresponds to r_1 and corresponding binary codeword 011_2 in Table 9.1, 119 corresponds to r_2 and code 1_2, and 123 and 168 correspond to 010_2 and 00_2, respectively. ■

9.2.1 Huffman Codes

When coding the gray levels of an image or the output of a gray-level mapping operation (pixel differences, run-lengths, and so on), *Huffman codes* contain the smallest possible number of code symbols (e.g., bits) per source symbol (e.g., gray-level value) subject to the constraint that the source symbols are coded *one at a time*.

The first step in Huffman's approach is to create a series of source reductions by ordering the probabilities of the symbols under consideration and combining the lowest probability symbols into a single symbol that replaces them in the next source reduction. Figure 9.2(a) illustrates the process for the gray-level distribution in Table 9.1. At the far left, the initial set of source symbols and their probabilities are ordered from top to bottom in terms of decreasing probability values. To form the first source reduction, the bottom two probabilities, 0.125 and 0.1875, are combined to form a "compound symbol" with probability 0.3125. This compound symbol and its associated probability are placed in the first source reduction column so that the probabilities of the reduced source are also ordered from the most to the least probable. This process is then repeated until a reduced source with two symbols (at the far right) is reached.

a
b

FIGURE 9.2
Huffman (a) source reduction and (b) code assignment procedures.

Original Source		Source Reduction	
Symbol	Probability	1	2
a_2	0.5	0.5	0.5
a_4	0.1875	0.3125	0.5
a_1	0.1875	0.1875	
a_3	0.125		

Original Source			Source Reduction			
Symbol	Probability	Code	1		2	
a_2	0.5	1	0.5	1	0.5	1
a_4	0.1875	00	0.3125	01	0.5	0
a_1	0.1875	011	0.1875	00		
a_3	0.125	010				

The second step in Huffman's procedure is to code each reduced source, starting with the smallest source and working back to the original source. The minimal length binary code for a two-symbol source, of course, consists of the symbols 0 and 1. As Fig. 9.2(b) shows, these symbols are assigned to the two symbols on the right (the assignment is arbitrary; reversing the order of the 0 and 1 would work just as well). As the reduced source symbol with probability 0.5 was generated by combining two symbols in the reduced source to its left, the 0 used to code it is now assigned to *both* of these symbols, and a 0 and 1 are arbitrarily appended to each to distinguish them from each other. This operation is then repeated for each reduced source until the original source is reached. The final code appears at the far left (column 3) in Fig. 9.2(b).

The Huffman code in Fig. 9.2(b) (and Table 9.1) is an instantaneous uniquely decodable block code. It is a *block code* because each source symbol is mapped into a fixed sequence of code symbols. It is *instantaneous* because each code word in a string of code symbols can be decoded without referencing succeeding symbols. That is, in any given Huffman code, no code word is a prefix of any other code word. And it is *uniquely decodable* because a string of code symbols can be decoded in only one way. Thus, any string of Huffman encoded symbols can be decoded by examining the individual symbols of the string in a left-to-right manner. For the 4×4 image in Example 9.1, a top-to-bottom left-to-right encoding based on the Huffman code in Fig. 9.2(b) yields the 29-bit string 10101011010110110000011110011. Because we are using an instantaneous uniquely decodable block code, there is no need to insert delimiters between the encoded pixels. A left-to-right scan of the resulting string reveals that the first valid code word is 1, which is the code for symbol a_2 or gray level 119. The next valid code word is 010, which corresponds to gray level 123. Con-

tinuing in this manner, we eventually obtain a completely decoded image that is equivalent to f in the example.

The source reduction and code assignment procedures just described are implemented by the following M-function, which we call huffman:

huffman

```
function CODE = huffman(p)
%HUFFMAN Builds a variable-length Huffman code for symbol source.
%   CODE = HUFFMAN(P) returns a Huffman code as binary strings in
%   cell array CODE for input symbol probability vector P. Each word
%   in CODE corresponds to a symbol whose probability is at the
%   corresponding index of P.
%
%   Based on huffman5 by Sean Danaher, University of Northumbria,
%   Newcastle UK. Available at the MATLAB Central File Exchange:
%   Category General DSP in Signal Processing and Communications.

% Check the input arguments for reasonableness.
error(nargchk(1, 1, nargin));
if (ndims(p) ~= 2) || (min(size(p)) > 1) || ~isreal(p) ...
        || ~isnumeric(p)
   error('P must be a real numeric vector.');
end

% Global variable surviving all recursions of function 'makecode'
global CODE
CODE = cell(length(p), 1);  % Init the global cell array

if length(p) > 1            % When more than one symbol ...
   p = p / sum(p);          % Normalize the input probabilities
   s = reduce(p);           % Do Huffman source symbol reductions
   makecode(s, []);         % Recursively generate the code
else
   CODE = {'1'};            % Else, trivial one symbol case!
end;

%--------------------------------------------------------------------%
function s = reduce(p)
% Create a Huffman source reduction tree in a MATLAB cell structure
% by performing source symbol reductions until there are only two
% reduced symbols remaining

s = cell(length(p), 1);

% Generate a starting tree with symbol nodes 1, 2, 3, ... to
% reference the symbol probabilities.
for i = 1:length(p)
   s{i} = i;
end

while numel(s) > 2
   [p, i] = sort(p);    % Sort the symbol probabilities
   p(2) = p(1) + p(2);  % Merge the 2 lowest probabilities
   p(1) = [];           % and prune the lowest one
```

```
    s = s(i);              % Reorder tree for new probabilities
    s{2} = {s{1}, s{2}}; % and merge & prune its nodes
    s(1) = [];             % to match the probabilities
end

%-------------------------------------------------------------------%
function makecode(sc, codeword)
% Scan the nodes of a Huffman source reduction tree recursively to
% generate the indicated variable length code words.

% Global variable surviving all recursive calls
global CODE

if isa(sc, 'cell')                  % For cell array nodes,
   makecode(sc{1}, [codeword 0]);   % add a 0 if the 1st element
   makecode(sc{2}, [codeword 1]);   % or a 1 if the 2nd
else                                % For leaf (numeric) nodes,
   CODE{sc} = char('0' + codeword); % create a char code string

end
```

The following command line sequence uses `huffman` to generate the code in Fig. 9.2:

```
>> p = [0.1875 0.5 0.125 0.1875];
>> c = huffman(p)
c =
    '011'
    '1'
    '010'
    '00'
```

Note that the output is a variable-length character array in which each row is a string of 0s and 1s—the binary code of the correspondingly indexed symbol in `p`. For example, `'010'` (at array index 3) is the code for the gray level with probability 0.125.

In the opening lines of huffman, input argument `p` (the input symbol probability vector of the symbols to be encoded) is checked for reasonableness and *global variable* `CODE` is initialized as a MATLAB cell array (defined in Section 2.10.6) with `length(p)` rows and a single column. All MATLAB global variables must be declared in the functions that reference them using a statement of the form




```
global X Y Z
```

This statement makes variables `X`, `Y`, and `Z` available to the function in which they are declared. When several functions declare the same global variable, they share a single copy of that variable. In `huffman`, the main routine and internal function `makecode` share global variable `CODE`. Note that it is customary to capitalize the names of global variables. Nonglobal variables are *local*

variables and are available only to the functions in which they are defined (not to other functions or the base workspace); they are typically denoted in lowercase.

In `huffman`, `CODE` is initialized using the `cell` function, whose syntax is

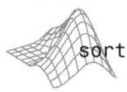
```
X = cell(m, n)
```



It creates an $m \times n$ array of empty matrices that can be referenced by cell or by content. Parentheses, "`()`", are used for *cell indexing*; curly braces, "`{}`", are used for *content indexing*. Thus, `X(1) = []` indexes and removes element 1 from the cell array, while `X{1} = []` sets the first cell array element to the empty matrix. That is, `X{1}` refers to the contents of the first element (an array) of `X`; `X(1)` refers to the element itself (rather than its content). Since cell arrays can be nested within other cell arrays, the syntax `X{1}{2}` refers to the content of the second element of the cell array that is in the first element of cell array `X`.

After `CODE` is initialized and the input probability vector is normalized [in the `p = p / sum(p)` statement], the Huffman code for normalized probability vector `p` is created in two steps. The first step, which is initiated by the `s = reduce(p)` statement of the main routine, is to call internal function `reduce`, whose job is to perform the source reductions illustrated in Fig. 9.2(a). In `reduce`, the elements of an initially empty source reduction cell array `s`, which is sized to match `CODE`, are initialized to their indices. That is, `s{1} = 1, s{2} = 2`, and so on. The cell equivalent of a binary tree for the source reductions is then created in the `while numel(s) > 2` loop. In each iteration of the loop, vector `p` is sorted in ascending order of probability. This is done by the `sort` function, whose general syntax is

```
[y, i] = sort(x)
```



where output `y` is the sorted elements of `x` and index vector `i` is such that `y = x(i)`. When `p` has been sorted, the lowest two probabilities are merged by placing their composite probability in `p(2)`, and `p(1)` is pruned. The source reduction cell array is then reordered to match `p` based on index `vector i` using `s = s(i)`. Finally, `s{2}` is replaced with a two-element cell array containing the merged probability indices via `s{2} = {s{1}, s{2}}` (an example of content indexing), and cell indexing is employed to prune the first of the two merged elements, `s(1)`, via `s(1) = []`. The process is repeated until only two elements remain in `s`.

Figure 9.3 shows the final output of the process for the symbol probabilities in Table 9.1 and Fig. 9.2(a). Figures 9.3(b) and (c) were generated by inserting

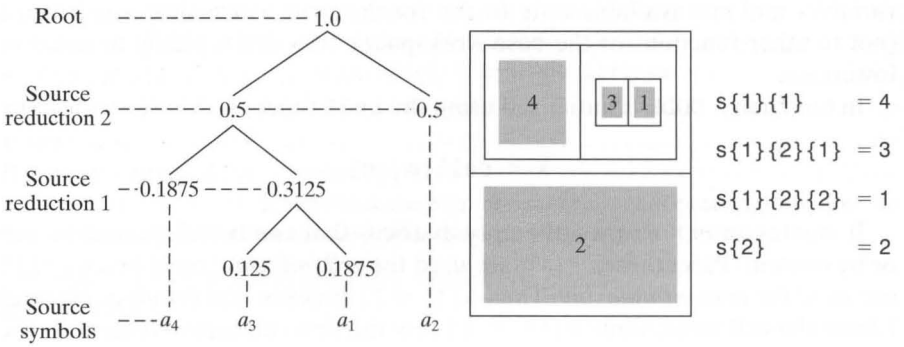
```
celldisp(s);
cellplot(s);
```



between the last two statements of the `huffman` main routine. MATLAB function `celldisp` prints a cell array's contents recursively; function `cell-`

a b c

FIGURE 9.3
Source reductions of Fig. 9.2(a) using function `huffman`: (a) binary tree equivalent; (b) display generated by `cellplot(s);` (c) `celldisp(s)` output.



`plot` produces a graphical depiction of a cell array as nested boxes. Note the one-to-one correspondence between the cell array elements in Fig. 9.3(b) and the source reduction tree nodes in Fig. 9.3(a): (1) Each two-way branch in the tree (which represents a source reduction) corresponds to a two-element cell array in `s`, and (2) each two-element cell array contains the indices of the symbols that were merged in the corresponding source reduction. For example, the merging of symbols a_3 and a_1 at the bottom of the tree produces the two-element cell array `s{1}{2}`, where `s{1}{2}{1} = 3` and `s{1}{2}{2} = 1` (the indices of symbol a_3 and a_1, respectively). The root of the tree is the top-level two-element cell array `s`.

The final step of the code generation process (i.e., the assignment of codes based on source reduction cell array `s`) is triggered by the final statement of `huffman`—the `makecode(s, [])` call. This call initiates a *recursive* code assignment process based on the procedure in Fig. 9.2(b). Although recursion generally provides no savings in storage (since a stack of values being processed must be maintained somewhere) or increase in speed, it has the advantage that the code is more compact and often easier to understand, particularly when dealing with recursively defined data structures like trees. Any MATLAB function can be used recursively; that is, it can call itself either directly or indirectly. When recursion is used, each function call generates a fresh set of local variables, independent of all previous sets.

Internal function `makecode` accepts two inputs: `codeword`, an array of 0s and 1s, and `sc`, a source reduction cell array element. When `sc` is itself a cell array, it contains the two source symbols (or composite symbols) that were joined during the source reduction process. Since they must be individually coded, a pair of recursive calls (to `makecode`) is issued for the elements—along with two appropriately updated code words (a 0 and 1 are appended to input `codeword`). When `sc` does not contain a cell array, it is the index of an original source symbol and is assigned a binary string created from input `codeword` using `CODE{sc} = char('0' + codeword)`. As was noted in Section 2.10.5, MATLAB function `char` converts an array containing positive integers that represent character codes into a MATLAB character array (the first 127 codes are ASCII). Thus, for example, `char('0' + [0 1 0])` produces the character

string '010', since adding a 0 to the ASCII code for a 0 yields an ASCII '0', while adding a 1 to an ASCII '0' yields the ASCII code for a 1, namely '1'.

Table 9.2 details the sequence of makecode calls that results for the source reduction cell array in Fig. 9.3. Seven calls are required to encode the four symbols of the source. The first call (row 1 of Table 9.2) is made from the main routine of huffman and launches the encoding process with inputs codeword and sc set to the empty matrix and cell array s, respectively. In accordance with standard MATLAB notation, {1x2 cell} denotes a cell array with one row and two columns. Since sc is almost always a cell array on the first call (the exception is a single symbol source), two recursive calls (see rows 2 and 7 of the table) are issued. The first of these calls initiates two more calls (rows 3 and 4) and the second of these initiates two additional calls (rows 5 and 6). Anytime that sc is not a cell array, as in rows 3, 5, 6, and 7 of the table, additional recursions are unnecessary; a code string is created from codeword and assigned to the source symbol whose index was passed as sc.

9.2.2 Huffman Encoding

Huffman code generation is not (in and of itself) compression. To realize the compression that is built into a Huffman code, the symbols for which the code was created, whether they are gray levels, run lengths, or the output of some other gray-level mapping operation, must be transformed or mapped (i.e., encoded) in accordance with the generated code.

EXAMPLE 9.2: Variable-length code mappings in MATLAB.

■ Consider the simple 16-byte 4×4 image:

```
>> f2 = uint8([2 3 4 2; 3 2 4 4; 2 2 1 2; 1 1 2 2])
f2 =
     2     3     4     2
     3     2     4     4
     2     2     1     2
     1     1     2     2
>> whos('f2')
  Name      Size      Bytes     Class Attributes
  f2        4x4       16        uint8
```

TABLE 9.2 Code assignment process for the source reduction cell array in Fig. 9.3.

Call	Origin	sc	codeword
1	main routine	{1x2 cell} [2]	[]
2	makecode	[4] {1x2 cell}	0
3	makecode	4	0 0
4	makecode	[3] [1]	0 1
5	makecode	3	0 1 0
6	makecode	1	0 1 1
7	makecode	2	1

Each pixel in `f2` is an 8-bit byte; 16 bytes are used to represent the entire image. Because the gray levels of `f2` are not equiprobable, a variable-length code (as was indicated in the last section) will reduce the amount of memory required to represent the image. Function `huffman` computes one such code:

```
>> c = huffman(hist(double(f2(:)), 4))
c =
    '011'
    '1'
    '010'
    '00'
```

Since Huffman codes are based on the relative frequency of occurrence of the source symbols being coded (not the symbols themselves), `c` is identical to the code that was constructed for the image in Example 9.1. In fact, image `f2` can be obtained from `f` in Example 9.1 by mapping gray levels 107, 119, 123, and 168 to 1, 2, 3, and 4, respectively. For either image, `p = [0.1875 0.5 0.125 0.1875]`.

A simple way to encode `f2` based on code `c` is to perform a straightforward lookup operation:

```
>> h1f2 = c(f2(:))'
h1f2 =
    Columns 1 through 9
    '1'   '010'   '1'   '011'   '010'   '1'   '1'   '011'   '00'
    Columns 10 through 16
    '00'   '011'   '1'   '1'   '00'   '1'   '1'
>> whos('h1f2')
    Name      Size      Bytes    Class Attributes
    h1f2      1x16      1018     cell
```

Here, `f2` (a two-dimensional array of class UINT8) is transformed into a cell array, `h1f2` (the transpose compacts the display). The elements of `h1f2` are strings of varying length and correspond to the pixels of `f2` in a top-to-bottom left-to-right (i.e., columnwise) scan. As can be seen, the encoded image uses 1018 bytes of storage—more than 60 times the memory required by `f2`!

The use of a cell array for `h1f2` is logical because it is one of two standard MATLAB data structures (see Section 2.10.6) for dealing with arrays of dissimilar data. In the case of `h1f2`, the dissimilarity is the length of the character strings and the price paid for transparently handling it via the cell array is the memory overhead (inherent in the cell array) that is required to track the position of the variable-length elements. We can eliminate this overhead by transforming `h1f2` into a conventional two-dimensional character array:

```
>> h2f2 = char(h1f2)'
```

```
h2f2 =
    1010011000011011
     1 11  1001  0
     0 10  1  1
>> whos('h2f2')
    Name      Size      Bytes    Class Attributes
    h2f2      3x16      96       char
```

Here, cell array `h1f2` is transformed into a 3×16 character array, `h2f2`. Each column of `h2f2` corresponds to a pixel of `f2` in a top-to-bottom left-to-right (i.e., columnwise) scan. Note that blanks are inserted to size the array properly and, since two bytes are required for each `'0'` or `'1'` of a code word, the total memory used by `h2f2` is 96 bytes—still six times greater than the original 16 bytes needed for `f2`. We can eliminate the inserted blanks using

```
>> h2f2 = h2f2(:);
>> h2f2(h2f2 == ' ') = [];
>> whos('h2f2')
    Name      Size      Bytes    Class Attributes
    h2f2      29x1      58       char
```

but the required memory is still greater than `f2`'s original 16 bytes.

To compress `f2`, code `c` must be applied at the bit level, with several encoded pixels packed into a single byte:

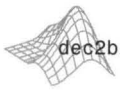
```
>> h3f2 = mat2huff(f2)
h3f2 =
    size: [4 4]
     min: 32769
    hist: [3 8 2 3]
    code: [43867 1944]
>> whos('h3f2')
    Name      Size      Bytes    Class Attributes
    h3f2      1x1       518      struct
```

Function `mat2huff` is described on the following page.

Although function `mat2huff` returns a structure, `h3f2`, requiring 518 bytes of memory, most of it is associated with either (1) structure variable overhead (recall from the Section 9.1 discussion of `imratio` that MATLAB uses 124 bytes of overhead per structure field) or (2) `mat2huff` generated information to facilitate future decoding. Neglecting this overhead, which is negligible when considering practical (i.e., normal size) images, `mat2huff` compresses `f2` by a factor of 4 : 1. The 16 8-bit pixels of `f2` are compressed into two 16-bit words—the elements in field `code` of `h3f2`:

```
>> hcode = h3f2.code;
>> whos('hcode')
```

```
    Name      Size      Bytes     Class Attributes
    hcode     1x2       4         uint16
```



Converts a decimal integer to a binary string. For more details, type `>>help dec2bin`.

```
>> dec2bin(double(hcode))
ans =
    1010101101011011
    0000011110011000
```

Note that `dec2bin` has been employed to display the individual bits of `h3f2.code`. Neglecting the terminating modulo-16 pad bits (i.e., the final three 0s), the 32-bit encoding is equivalent to the previously generated (see Section 9.2.1) 29-bit instantaneous uniquely decodable block code, 10101011010110110000011110011. ■

As was noted in the preceding example, function `mat2huff` embeds the information needed to decode an encoded input array (e.g., its original dimensions and symbol probabilities) in a single MATLAB structure variable. The information in this structure is documented in the help text section of `mat2huff` itself:

`mat2huff`

```
function y = mat2huff(x)
%MAT2HUFF Huffman encodes a matrix.
%   Y = MAT2HUFF(X) Huffman encodes matrix X using symbol
%   probabilities in unit-width histogram bins between X's minimum
%   and maximum values. The encoded data is returned as a structure
%   Y:
%      Y.code   The Huffman-encoded values of X, stored in
%               a uint16 vector.  The other fields of Y contain
%               additional decoding information, including:
%      Y.min    The minimum value of X plus 32768
%      Y.size   The size of X
%      Y.hist   The histogram of X
%
%   If X is logical, uint8, uint16, uint32, int8, int16, or double,
%   with integer values, it can be input directly to MAT2HUFF. The
%   minimum value of X must be representable as an int16.
%
%   If X is double with non-integer values---for example, an image
%   with values between 0 and 1---first scale X to an appropriate
%   integer range before the call. For example, use Y =
%   MAT2HUFF(255*X) for 256 gray level encoding.
%
%   NOTE: The number of Huffman code words is round(max(X(:))) -
%   round(min(X(:))) + 1.  You may need to scale input X to generate
%   codes of reasonable length.  The maximum row or column dimension
%   of X is 65535.
%
%   See also HUFF2MAT.
```

```
if ndims(x) ~= 2 || ~isreal(x) || (~isnumeric(x) && ~islogical(x))
   error('X must be a 2-D real numeric or logical matrix.');
end

% Store the size of input x.
y.size = uint32(size(x));

% Find the range of x values and store its minimum value biased
% by +32768 as a UINT16.
x = round(double(x));
xmin = min(x(:));
xmax = max(x(:));
pmin = double(int16(xmin));
pmin = uint16(pmin + 32768);    y.min = pmin;

% Compute the input histogram between xmin and xmax with unit
% width bins, scale to UINT16, and store.
x = x(:)';
h = histc(x, xmin:xmax);
if max(h) > 65535
    h = 65535 * h / max(h);
end
h = uint16(h);   y.hist = h;

% Code the input matrix and store the result.
map = huffman(double(h));          % Make Huffman code map
hx = map(x(:) - xmin + 1);         % Map image
hx = char(hx)';                    % Convert to char array
hx = hx(:)';
hx(hx == ' ') = [];                % Remove blanks
ysize = ceil(length(hx) / 16);     % Compute encoded size
hx16 = repmat('0', 1, ysize * 16); % Pre-allocate modulo-16 vector
hx16(1:length(hx)) = hx;           % Make hx modulo-16 in length
hx16 = reshape(hx16, 16, ysize);   % Reshape to 16-character words
hx16 = hx16' - '0';                % Convert binary string to decimal
twos = pow2(15:-1:0);
y.code = uint16(sum(hx16 .* twos(ones(ysize, 1), :), 2))';
```



This function is similar to `hist`. For more details, type `>>help histc`.

Note that the statement `y = mat2huff(x)` Huffman encodes input matrix `x` using unit-width histogram bins between the minimum and maximum values of `x`. When the encoded data in `y.code` is later decoded, the Huffman code needed to decode it must be re-created from `y.min`, the minimum value of `x`, and `y.hist`, the histogram of `x`. Rather than preserving the Huffman code itself, `mat2huff` keeps the probability information needed to regenerate it. With this, and the original dimensions of matrix `x`, which is stored in `y.size`, function `huff2mat` of Section 9.2.3 (the next section) can decode `y.code` to reconstruct `x`.

The steps involved in the generation of `y.code` are summarized as follows:

1. Compute the histogram, `h`, of input `x` between the minimum and maximum values of `x` using unit-width bins and scale it to fit in a `uint16` vector.

2. Use `huffman` to create a Huffman code, called `map`, based on the scaled histogram, `h`.
3. Map input `x` using `map` (this creates a cell array) and convert it to a character array, `hx`, removing the blanks that are inserted like in `h2f2` of Example 9.2.
4. Construct a version of vector `hx` that arranges its characters into 16-character segments. This is done by creating a modulo-16 character vector that will hold it (`hx16` in the code), copying the elements of `hx` into it, and reshaping it into a 16 row by `ysize` array, where `ysize = ceil(length(hx) / 16)`. Recall from Section 4.2 that the `ceil` function rounds a number toward positive infinity. As mentioned in Section 8.3.1, the function

   ```
   y = reshape(x, m, n)
   ```

   returns an `m` by `n` matrix whose elements are taken column wise from `x`. An error is returned if `x` does not have `mn` elements.
5. Convert the 16-character elements of `hx16` to 16-bit binary numbers (i.e., `unit16`). Three statements are substituted for the more compact `y = uint16(bin2dec(hx16()`. They are the core of `bin2dec`, which returns the decimal equivalent of a binary string (e.g., `bin2dec('101')` returns 5) but are faster because of decreased generality. MATLAB function `pow2(y)` is used to return an array whose elements are 2 raised to the `y` power. That is, `twos = pow2(15: –1: 0)` creates the array [32768 16384 8192... 8 4 2 1].



EXAMPLE 9.3: Encoding with `mat2huff`.

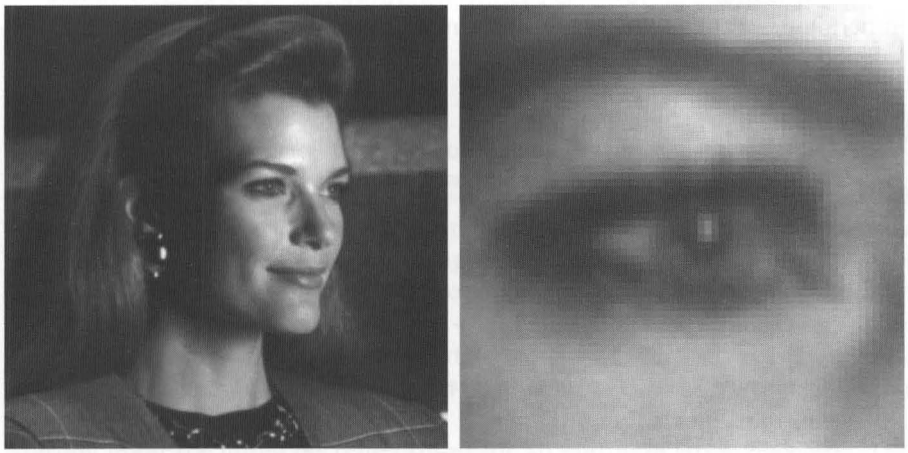
■ To illustrate further the compression performance of Huffman encoding, consider the 512 × 512 8-bit monochrome image of Fig. 9.4(a). The compression of this image using `mat2huff` is carried out by the following command sequence:

```
>> f = imread('Tracy.tif');
>> c = mat2huff(f);
>> cr1 = imratio(f, c)
cr1 =
    1.2191
```

By removing the coding redundancy associated with its conventional 8-bit binary encoding, the image has been compressed to about 80% of its original size (even with the inclusion of the decoding overhead information).

Because the output of `mat2huff` is a structure, we write it to disk using the `save` function:

```
>> save SqueezeTracy c;
>> cr2 = imratio('Tracy.tif', 'SqueezeTracy.mat')
cr2 =
    1.2365
```



a b

FIGURE 9.4 An 8-bit monochrome image of a woman and a closeup of her right eye.

The save function, like the Save Workspace As and Save Selection As menu commands in Section 1.7.4, appends a `.mat` extension to the file that is created. The resulting file—in this case, `SqueezeTracy.mat`, is called a *MAT-file*. It is a binary data file containing workspace variable names and values. Here, it contains the single workspace variable `c`. Finally, we note that the small difference in compression ratios `cr1` and `cr2` computed previously is due to MATLAB data file overhead. ■

9.2.3 Huffman Decoding

Huffman encoded images are of little use unless they can be decoded to re-create the original images from which they were derived. For output `y = mat2huff(x)` of the previous section, the decoder must first compute the Huffman code used to encode `x` (based on its histogram and related information in `y`) and then inverse map the encoded data (also extracted from `y`) to rebuild `x`. As can be seen in the following listing of function `x = huff2mat(y)`, this process can be broken into five basic steps:

1. Extract dimensions `m` and `n`, and minimum value `xmin` (of eventual output `x`) from input structure `y`.
2. Re-create the Huffman code that was used to encode `x` by passing its histogram to function `huffman`. The generated code is called `map` in the listing.
3. Build a data structure (transition and output table `link`) to streamline the decoding of the encoded data in `y.code` through a series of computationally efficient binary searches.
4. Pass the data structure and the encoded data [i.e., `link` and `y.code`] to C function `unravel`. This function minimizes the time required to perform the binary searches, creating decoded output vector `x` of class `double`.
5. Add `xmin` to each element of `x` and reshape it to match the dimensions of the original `x` (i.e., `m` rows and `n` columns).

A unique feature of `huff2mat` is the incorporation of MATLAB callable C function `unravel` (see Step 4), which makes the decoding of most normal resolution images nearly instantaneous.

`huff2mat`

```
function x = huff2mat (y)
%HUFF2MAT Decodes a Huffman encoded matrix.
%   X = HUFF2MAT(Y) decodes a Huffman encoded structure Y with uint16
%   fields:
%      Y.min    Minimum value of X plus 32768
%      Y.size   Size of X
%      Y.hist   Histogram of X
%      Y.code   Huffman code
%
%   The output X is of class double.
%
%   See also MAT2HUFF.

if ~isstruct(y) || ~isfield(y, 'min') || ~isfield(y, 'size') || ...
       ~isfield(y, 'hist') || ~isfield(y, 'code')
   error('The input must be a structure as returned by MAT2HUFF.');
end

sz = double(y.size);   m = sz(1);   n = sz(2);
xmin = double(y.min) - 32768;            % Get X minimum
map = huffman(double(y.hist));           % Get Huffman code (cell)

% Create a binary search table for the Huffman decoding process.
% 'code' contains source symbol strings corresponding to 'link'
% nodes, while 'link' contains the addresses (+) to node pairs for
% node symbol strings plus '0' and '1' or addresses (-) to decoded
% Huffman codewords in 'map'. Array 'left' is a list of nodes yet to
% be processed for 'link' entries.

code = cellstr(char('', '0', '1'));     % Set starting conditions as
link = [2; 0; 0];   left = [2 3];       % 3 nodes w/2 unprocessed
found = 0;   tofind = length(map);      % Tracking variables

while ~isempty(left) && (found < tofind)
   look = find(strcmp(map, code{left(1)}));    % Is string in map?
   if look                                % Yes
      link(left(1)) = -look;              % Point to Huffman map
      left = left(2:end);                 % Delete current node
      found = found + 1;                  % Increment codes found

   else                                   % No, add 2 nodes & pointers
      len = length(code);                 % Put pointers in node
      link(left(1)) = len + 1;

      link = [link; 0; 0];                % Add unprocessed nodes
      code{end + 1} = strcat(code{left(1)}, '0');
      code{end + 1} = strcat(code{left(1)}, '1');

      left = left(2:end);                 % Remove processed node
      left = [left len + 1 len + 2];  % Add 2 unprocessed nodes
```

```
    end
end

x = unravel(y.code', link, m * n);     % Decode using C 'unravel'
x = x + xmin - 1;                      % X minimum offset adjust
x = reshape(x, m, n);                  % Make vector an array
```

As indicated earlier, `huff2mat`-based decoding is built on a series of binary searches or two-outcome decoding decisions. Each element of a sequentially scanned Huffman encoded string—which must of course be a `'0'` or a `'1'`—triggers a binary decoding decision based on transition and output table `link`. The construction of `link` begins with its initialization in statement `link = [2; 0; 0]`. Each element in the starting three-state `link` array corresponds to a Huffman encoded binary string in the corresponding cell array `code`; that is, `code = cellstr(char(', '0', '1'))`. The null string, `code(1)`, is the starting point (or initial decoding state) for all Huffman string decoding. The associated 2 in `link(1)` identifies the two possible decoding states that follow from appending a `'0'` and `'1'` to the null string. If the next encountered Huffman encoded bit is a `'0'`, the next decoding state is `link(2)` [since `code(2) = '0'`, the null string concatenated with `'0'`]; if it is a `'1'`, the new state is `link(3)` (at index (2 + 1) or 3, with `code(3) = '1'`). Note that the corresponding `link` array entries are 0—indicating that they have not yet been processed to reflect the proper decisions for Huffman code `map`. During the construction of `link`, if either string (i.e., the `'0'` or `'1'`) is found in `map` (i.e., it is a valid Huffman code word), the corresponding 0 in `link` is replaced by the negative of the corresponding `map` index (which is the decoded value). Otherwise, a new (positive valued) link index is inserted to point to the two new states (possible Huffman code words) that logically follow (either `'00'` and `'01'` or `'10'` and `'11'`). These new and as yet unprocessed `link` elements expand the size of `link` (cell array `code` must also be updated), and the construction process is continued until there are no unprocessed elements left in `link`. Rather than continually scanning `link` for unprocessed elements, however, `huff2mat` maintains a tracking array, called `left`, which is initialized to `[2, 3]` and updated to contain the indices of the link elements that have not been examined.

Table 9.3 shows the `link` table that is generated `for` the Huffman code in Example 9.2. If each `link` index is viewed as a decoding state, `i`, each binary coding decision (in a left-to-right scan of an encoded string) and/or Huffman decoded output is determined by `link(i)`:

1. If `link(i) < 0` (i.e., negative), a Huffman code word has been decoded. The decoded output is `|link(i)|`, where `| |` denotes the absolute value.
2. If `link(i) > 0` (i.e., positive) and the next encoded bit to be processed is a 0, the next decoding state is index `link (i)`. That is, we let `i = link(i)`.
3. If `link(i) > 0` and the next encoded bit to be processed is a 1, the next decoding state is index `link(i) + 1`. That is, `i = link(i) + 1`.

As noted previously; positive `link` entries correspond to binary decoding transitions, while negative entries determine decoded output values. As each

TABLE 9.3
Decoding table for the source reduction cell array in Fig. 9.3.

Index `i`	Value in `link(i)`
1	2
2	4
3	-2
4	-4
5	6
6	-3
7	-1

Huffman code word is decoded, a new binary search is started at `link` index `i = 1`. For encoded string 101010110101 of Example 9.2, the resulting state transition sequence is `i = 1, 3, 1, 2, 5, 6, 1, ...`; the corresponding output sequence is `–, |–2|, –, –, –, |–3|, –, ...`, where – is used to denote the absence of an output. Decoded output values 2 and 3 are the first two pixels of the first line of test image `f2` in Example 9.2.

C function `unravel` accepts the link structure just described and uses it to drive the binary searches required to decode input `hx`. Figure 9.5 diagrams its basic operation, which follows the decision-making process that was described in conjunction with Table 9.3. Note, however, that modifications are needed to compensate for the fact that C arrays are indexed from 0 rather than 1.

Both C and Fortran functions can be incorporated into MATLAB and serve two main purposes: (1) They allow large preexisting C and Fortran programs to be called from MATLAB without having to be rewritten as M-files, and (2) they streamline bottleneck computations that do not run fast enough as MATLAB M-files but can be coded in C or Fortran for increased efficiency. Whether C or Fortran is used, the resulting functions are referred to as *MEX-files*; they behave as though they are M-files or ordinary MATLAB functions. Unlike M-files, however, they must be compiled and linked using MATLAB's `mex` script before they can be called. To compile and link `unravel` on a Windows platform from the MATLAB command line prompt, for example, we type



A MATLAB external function produced from C or Fortran code. It has a platform-dependent extension (e.g., `.mexw32` for Windows).

```
>> mex unravel.c
```

A MEX-file named `unravel.mexw32` with extension `.mexw32` will be created. Any help text, if desired, must be provided as a separate M-file of the same name (it will have a `.m` extension).



The C source code used to build a MEX-file..

The source code for *C MEX-file* `unravel` has a `.c` extension and as follows:

`unravel.c`

```
/*=================================================================
* unravel.c
* Decodes a variable length coded bit sequence (a vector of
* 16-bit integers) using a binary sort from the MSB to the LSB
* (across word boundaries) based on a transition table.
*=================================================================*/
#include "mex.h"
```

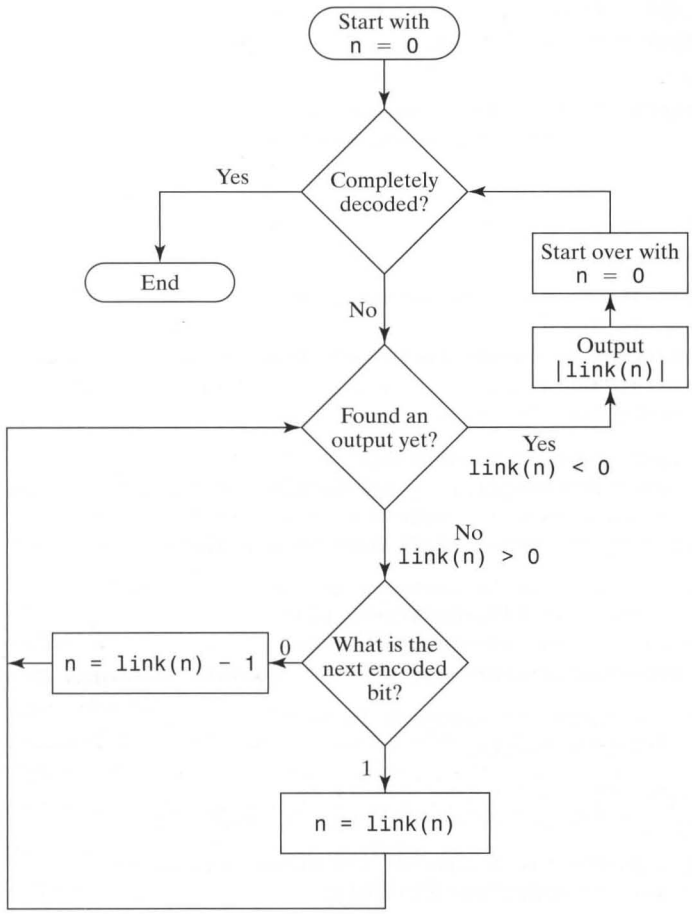


FIGURE 9.5
Flow diagram for C function `unravel`.

```
void unravel(uint16_T *hx, double *link, double *x,
    double xsz, int hxsz)
{
   int i = 15, j = 0, k = 0, n = 0;     /* Start at root node, 1st */
                                        /* hx bit and x element */
while (xsz - k)   {                     /* Do until x is filled */
   if (*(link + n) > 0)   {             /* Is there a link? */
      if ((*(hx + j) >> i) & 0x0001)    /* Is bit a 1? */
          n = *(link + n);              /* Yes, get new node */
      else n = *(link + n) - 1;         /* It's 0 so get new node */
      if (i) i--; else {j++; i = 15;} /* Set i, j to next bit */
      if (j > hxsz)                     /* Bits left to decode? */
         mexErrMsgTxt("Out of code bits ???");
      }
else    {                               /* It must be a leaf node */
   *(x + k++) = - *(link + n);          /* Output value */
    n = 0;    }                         /* Start over at root */
}
```

```
if (k == xsz - 1)                          /* Is one left over? */
  *(x + k++) = - *(link + n);
}

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
   double *link, *x, xsz;
   uint16_T *hx;
   int hxsz;

   /* Check inputs for reasonableness */
   if (nrhs != 3)
      mexErrMsgTxt("Three inputs required.");
   else if (nlhs > 1)
      mexErrMsgTxt("Too many output arguments.");

   /* Is last input argument a scalar? */
   if(!mxIsDouble(prhs[2])  || mxIsComplex(prhs[2])  ||
         mxGetN(prhs[2]) * mxGetM(prhs[2]) != 1)
      mexErrMsgTxt("Input XSIZE must be a scalar.");

   /* Create input matrix pointers and get scalar */
   hx = (uint16_T *) mxGetData(prhs[0]);
   link = (double *) mxGetData(prhs[1]);
   xsz = mxGetScalar(prhs[2]);          /* returns DOUBLE */

   /* Get the number of elements in hx */
   hxsz = mxGetM(prhs[0]);

   /* Create 'xsz' x 1 output matrix */
   plhs[0] = mxCreateDoubleMatrix(xsz, 1, mxREAL);

   /* Get C pointer to a copy of the output matrix */
   x = (double *) mxGetData(plhs[0]);

   /* Call the C subroutine */
   unravel(hx, link, x, xsz, hxsz);

}
```

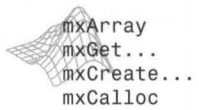
The companion help text is provided in M-file `unravel.m`:

unravel.m

```
%UNRAVEL Decodes a variable-length bit stream.
%   X = UNRAVEL(Y, LINK, XLEN) decodes UINT16 input vector Y based on
%   transition and output table LINK. The elements of Y are
%   considered to be a contiguous stream of encoded bits--i.e., the
%   MSB of one element follows the LSB of the previous element. Input
%   XLEN is the number code words in Y, and thus the size of output
%   vector X (class DOUBLE). Input LINK is a transition and output
%   table (that drives a series of binary searches):
%
%    1. LINK(0) is the entry point for decoding, i.e., state n = 0.
%    2. If LINK(n) < 0, the decoded output is |LINK(n)|; set n = 0.
%    3. If LINK(n) > 0, get the next encoded bit and transition to
%       state [LINK(n) - 1] if the bit is 0, else LINK(n).
```

Like all C MEX-files, C MEX-file `unravel.c` consists of two distinct parts: a *computational routine* and a *gateway routine*. The computational routine, also named `unravel`, contains the C code that implements the `link`-based decoding process of Fig. 9.5. The gateway routine, which must always be named `mexFunction`, interfaces C computational routine `unravel` to MATLAB. It uses MATLAB's standard MEX-file interface, which is based on the following:

1. Four standardized input/output parameters—`nlhs`, `plhs`, `nrhs`, and `prhs`. These parameters are the number of left-hand-side output arguments (an integer), an array of pointers to the left-hand-side output arguments (all MATLAB arrays), the number of right-hand-side input arguments (another integer), and an array of pointers to the right-hand-side input arguments (also MATLAB arrays), respectively.
2. A MATLAB provided set of *Application Program interface* (API) functions. API functions that are prefixed with `mx` are used to create, access, manipulate, and/or destroy structures of class `mxArray`. For example,



   - mxCalloc dynamically allocates memory like a standard C *calloc* function. Related functions include `mxMalloc` and `mxRealloc` that are used in place of the C *malloc* and *realloc* functions.
   - `mxGetScalar` extracts a scalar from input array `prhs`. Other `mxGet` functions, like `mxGetM`, `mxGetN`, and `mxGetString`, extract other types of data.
   - `mxCreateDoubleMatrix` creates a MATLAB output array for `plhs`. Other `mxCreate` functions, like `mxCreateString` and `mxCreateNumericArray`, facilitate the creation of other data types.

   API functions prefixed by `mex` perform operations in the MATLAB environment. For example, `mexErrMsgTxt` outputs a message to the MATLAB Command Window.



Function prototypes for the API `mex` and `mx` routines noted in item 2 of the preceding list are maintained in MATLAB header files `mex.h` and `matrix.h`, respectively. Both are located in the `<matlab>/extern/include` directory, where `<matlab>` denotes the top-level directory where MATLAB is installed on your system. Header `mex.h`, which must be included at the beginning of all MEX-files (note the C file inclusion statement `#include "mex.h "` at the start of MEX-file `unravel`), includes header file `matrix.h`. The prototypes of the `mex` and `mx` interface routines that are contained in these files define the parameters that they use and provide valuable clues about their general operation. Additional information is available in the *External Interfaces* section of the MATLAB documentation.

Figure 9.6 summarizes the preceding discussion, details the overall structure of C MEX-file `unravel`, and describes the flow of information between it and M-file `huff2mat`. Though constructed in the context of Huffman decoding, the concepts illustrated are easily extended to other C- and/or Fortran-based MATLAB functions.

EXAMPLE 9.4: Decoding with `huff2mat`.



Function `load` reads MATLAB variables from a file and loads them into the workspace. The variable names are maintained through a `save`/`load` sequence.

■ The Huffman encoded image of Example 9.3 can be decoded with the following sequence of commands:

```
>> load SqueezeTracy;
>> g = huff2mat(c);
>> f = imread('Tracy.tif');
>> rmse = compare(f, g)

rmse =
     0
```

Note that the overall encoding-decoding process is information preserving; the root-mean-square error between the original and decompressed images is 0. Because such a large part of the decoding job is done in C MEX-file `unravel`, `huff2mat` is slightly faster than its encoding counterpart, `mat2huff`. Note the use of the `load` function to retrieve the MAT-file encoded output from Example 9.2. ■

9.3 Spatial Redundancy

Consider the images shown in Figs. 9.7(a) and (c). As Figs. 9.7(b) and (d) show, they have virtually identical histograms. Note also that the histograms are trimodal, indicating the presence of three dominant ranges of gray-level values. Because the gray levels of the images are not equally probable, variable-length coding can be used to reduce the coding redundancy that would result from a natural binary coding of their pixels:

```
>> f1 = imread('Random Matches.tif');
>> c1 = mat2huff(f1);
>> ntrop(f1)

ans =
    7.4253

>> imratio(f1, c1)

ans =
    1.0704

>> f2 = imread('Aligned Matches.tif');
>> c2 = mat2huff(f2);
>> ntrop(f2)

ans =
    7.3505

>> imratio(f2, c2)

ans =
    1.0821
```

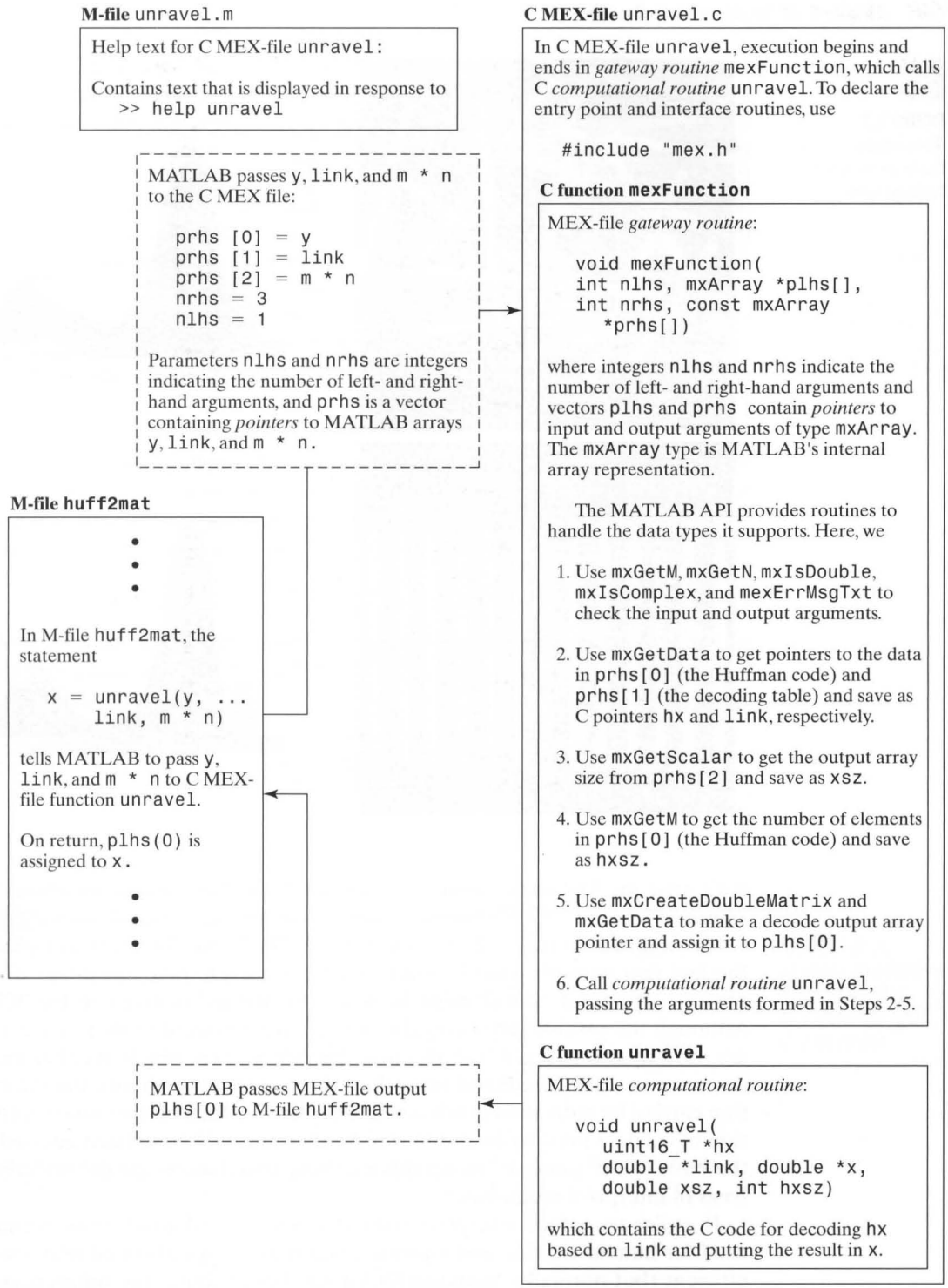
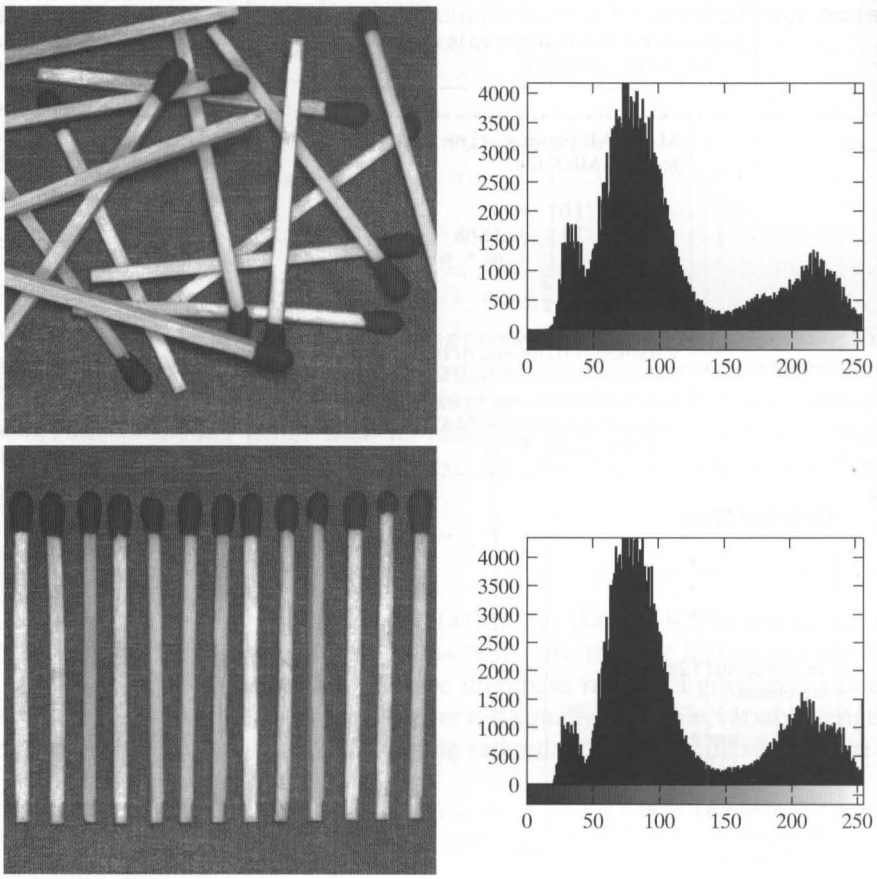


FIGURE 9.6 The interaction of M-file `huff2mat` and MATLAB callable C function `unravel`. Note that MEX-file `unravel` contains two functions: gateway routine `mexFunction` and computational routine `unravel`. Help text for MEX-file `unravel` is contained in the separate M-file, also named `unravel`.

a b
c d

FIGURE 9.7
Two images and their gray-level histograms.



Note that the first-order entropy estimates of the two images are about the same (7.4253 and 7.3505 bits/pixel); they are compressed similarly by `mat2huff` (with compression ratios of 1.0704 versus 1.0821). These observations highlight the fact that variable-length coding is not designed to take advantage of the obvious structural relationships between the aligned matches in Fig. 9.7(c). Although the pixel-to-pixel correlations are more evident in that image, they are present also in Fig. 9.7(a). Because the values of the pixels in either image can be reasonably predicted from the values of their neighbors, the information carried by individual pixels is relatively small. Much of the visual contribution of a single pixel to an image is redundant; it could have been guessed on the basis of the values of its neighbors. These correlations are the underlying basis of interpixel redundancy.

In order to reduce interpixel redundancies, the 2-D pixel array normally used for human viewing and interpretation must be transformed into a more efficient (but normally "nonvisual") format. For example, the differences between adjacent pixels can be used to represent an image. Transformations of this type (that is, those that remove interpixel redundancy) are referred to as

mappings. They are called *reversible mappings* if the original image elements can be reconstructed from the transformed data set.

A simple mapping procedure is illustrated in Fig. 9.8. The approach, called *lossless predictive coding*, eliminates the interpixel redundancies of closely spaced pixels by extracting and coding only the new information in each pixel. The *new information* of a pixel is defined as the difference between the actual and predicted value of that pixel. As can be seen, the system consists of an encoder and decoder, each containing an identical *predictor*. As each successive pixel of the input image, denoted f_n, is introduced to the encoder, the predictor generates the anticipated value of that pixel based on some number of past inputs. The output of the predictor is then rounded to the nearest integer, denoted $\hat{f}_n$, and used to form the difference or prediction error

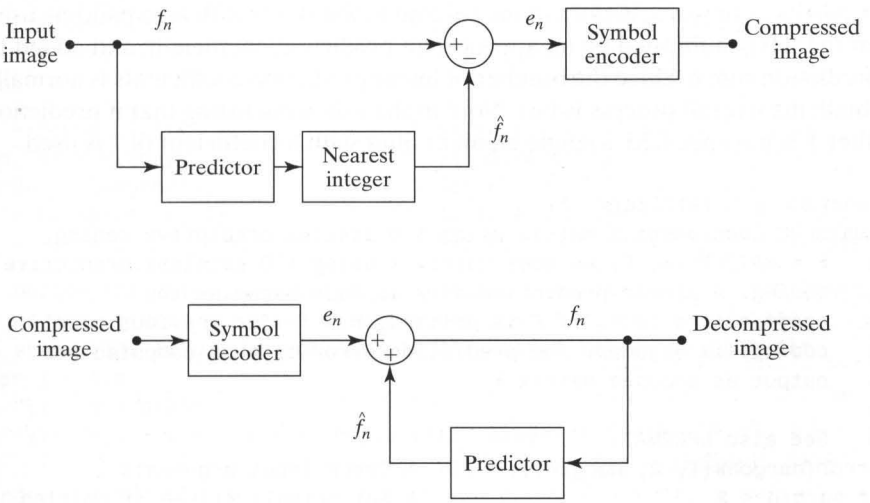
$$e_n = f_n - \hat{f}_n$$

which is coded using a variable-length code (by the symbol coder) to generate the next element of the compressed data stream. The decoder of Fig. 9.9(b) reconstructs e_n from the received variable-length code words and performs the inverse operation

$$f_n = e_n + \hat{f}_n$$

Various local, global, and adaptive methods can be used to generate $\hat{f}_n$. In most cases, however, the prediction is formed by a linear combination of m previous pixels. That is,

$$\hat{f}_n = \text{round}\left[\sum_{i=1}^{m} \alpha_i f_{n-i}\right]$$

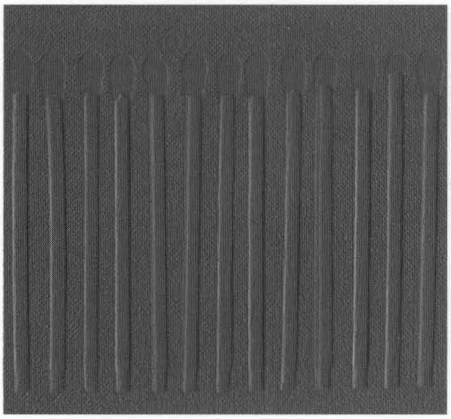


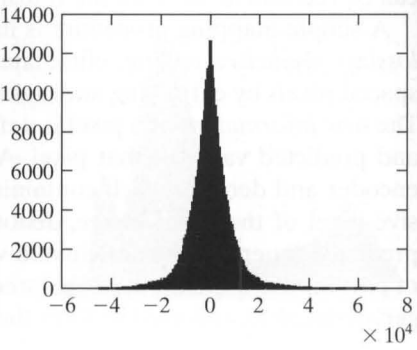
a
b

FIGURE 9.8 A lossless predictive coding model: (a) encoder and (b) decoder.

a b

FIGURE 9.9
(a) The prediction error image for Fig. 9.7(c) with `f=[1]`.
(b) Histogram of the prediction error.





where m is the order of the linear predictor, "round" is a function used to denote the rounding or nearest integer operation (like function `round` in MATLAB), and the α_i for $i = 1, 2, \ldots, m$ are prediction coefficients. For 1-D linear predictive coding, this equation can be rewritten

$$\hat{f}(x,y) = \text{round}\left[\sum_{i=1}^{m} \alpha_i f(x, y-i)\right]$$

where each subscripted variable is now expressed explicitly as a function of spatial coordinates x and y. Note that prediction $\hat{f}(x,y)$ is a function of the previous pixels on the current scan line alone.

M-functions `mat2lpc` and `lpc2mat` implement the predictive encoding and decoding processes just described (minus the symbol coding and decoding steps). Encoding function `mat2lpc` employs an `for` loop to build simultaneously the prediction of every pixel in input `x`. During each iteration, `xs`, which begins as a copy of `x`, is shifted one column to the right (with zero padding used on the left), multiplied by an appropriate prediction coefficient, and added to prediction sum `p`. Since the number of linear prediction coefficients is normally small, the overall process is fast. Note in the following listing that if prediction filter `f` is not specified, a single element filter with a coefficient of 1 is used.

`mat2lpc`

```
function y = mat2lpc(x, f)
%MAT2LPC Compresses a matrix using 1-D lossles predictive coding.
%   Y = MAT2LPC(X, F) encodes matrix X using 1-D lossless predictive
%   coding. A linear prediction of X is made based on the
%   coefficients in F. If F is omitted, F = 1 (for previous pixel
%   coding) is assumed. The prediction error is then computed and
%   output as encoded matrix Y.
%
%   See also LPC2MAT.
error(nargchk(1, 2, nargin));       % Check input arguments
if nargin < 2                       % Set default filter if omitted
    f = 1;
```

```
end

x = double(x);                          % Ensure double for computations
[m, n] = size(x);                       % Get dimensions of input matrix
p = zeros(m, n);                        % Init linear prediction to 0
xs = x;     zc = zeros(m, 1);           % Prepare for input shift and pad

for j = 1:length(f)                     % For each filter coefficient ...
    xs = [zc xs(:, 1:end - 1)];         % Shift and zero pad x
    p = p + f(j) * xs;                  % Form partial prediction sums
end

y = x - round(p);                       % Compute prediction error
```

Decoding function `lpc2mat` performs the inverse operations of encoding counterpart `mat2lpc`. As can be seen in the following listing, it employs an `n` iteration `for` loop, where `n` is the number of columns in encoded input matrix `y`. Each iteration computes only one column of decoded output `x`, since each decoded column is required for the computation of all subsequent columns. To decrease the time spent in the `for` loop, `x` is preallocated to its maximum padded size before starting the loop. Note also that the computations employed to generate predictions are done in the same order as they were in `lpc2mat` to avoid floating point round-off error.

lpc2mat

```
function x = lpc2mat(y, f)
%LPC2MAT Decompresses a 1-D lossless predictive encoded matrix.
%   X = LPC2MAT(Y, F) decodes input matrix Y based on linear
%   prediction coefficients in F and the assumption of 1-D lossless
%   predictive coding. If F is omitted, filter F = 1 (for previous
%   pixel coding) is assumed.
%
%   See also MAT2LPC.
error(nargchk(1, 2, nargin));     % Check input arguments
if nargin < 2                     % Set default filter if omitted
    f = 1;
end

f = f(end:-1:1);                  % Reverse the filter coefficients
[m, n] = size(y);                 % Get dimensions of output matrix
order = length(f);                % Get order of linear predictor
f = repmat(f, m, 1);              % Duplicate filter for vectorizing
x = zeros(m, n + order);          % Pad for 1st 'order' column decodes

% Decode the output one column at a time. Compute a prediction based
% on the 'order' previous elements and add it to the prediction
% error. The result is appended to the output matrix being built.
for j = 1:n
   jj = j + order;
   x(:, jj) = y(:, j) + round(sum(f(:, order:-1:1) .* ...
                              x(:, (jj - 1):-1:(jj - order)), 2));
end

x = x(:, order + 1:end);          % Remove left padding
```

EXAMPLE 9.5: Lossless predictive coding.

■ Consider encoding the image of Fig. 9.7(c) using the simple first-order linear predictor

$$\hat{f}(x,y) = \text{round}[\alpha f(x, y-1)]$$

A predictor of this form commonly is called a *previous pixel* predictor, and the corresponding predictive coding procedure is referred to as *differential coding* or *previous pixel coding*. Figure 9.9(a) shows the prediction error image that results with $\alpha = 1$. Here, gray level 128 corresponds to a prediction error of 0, while nonzero positive and negative errors (under- and overestimates) are scaled by `mat2gray` to become lighter or darker shades of gray, respectively:

```
>> f = imread('Aligned Matches.tif');
>> e = mat2lpc(f);
>> imshow(mat2gray(e));
>> ntrop(e)
ans =
    5.9727
```

Note that the entropy of the prediction error, `e`, is substantially lower than the entropy of the original image, `f`. The entropy has been reduced from the 7.3505 bits/pixel (computed at the beginning of this section) to 5.9727 bits/pixel, despite the fact that for m-bit images, $(m+1)$-bit numbers are needed to represent accurately the resulting error sequence. This reduction in entropy means that the prediction error image can be coded more efficiently that the original image—which, of course, is the goal of the mapping. Thus, we get

```
>> c = mat2huff(e);
>> cr = imratio(f, c)
cr =
    1.3311
```

and see that the compression ratio has, as expected, increased from 1.0821 (when Huffman coding the gray levels directly) to 1.3311.

The histogram of prediction error `e` is shown in Fig. 9.9(b)—and computed as follows:

```
>> [h, x] = hist(e(:) * 512, 512);
>> figure; bar(x, h, 'k');
```

Note that it is highly peaked around 0 and has a relatively small variance in comparison to the input image's gray-level distribution [see Fig. 9.7(d)]. This reflects, as did the entropy values computed earlier, the removal of a great deal of interpixel redundancy by the prediction and differencing process. We conclude the example by demonstrating the lossless nature of the predictive coding scheme—that is, by decoding `c` and comparing it to starting image `f`:

```
>> g = lpc2mat(huff2mat(c));
>> compare(f, g)
ans =
     0
```

■

9.4 Irrelevant Information

Unlike coding and interpixel redundancy, psychovisual redundancy is associated with real or quantifiable visual information. Its elimination is desirable because the information itself is not essential for normal visual processing. Since the elimination of psychovisually redundant data results in a loss of quantitative information, it is called *quantization*. This terminology is consistent with the normal usage of the word, which generally means the mapping of a broad range of input values to a limited number of output values. As it is an irreversible operation (i.e., visual information is lost), quantization results in lossy data compression.

EXAMPLE 9.6: Compression by quantization.

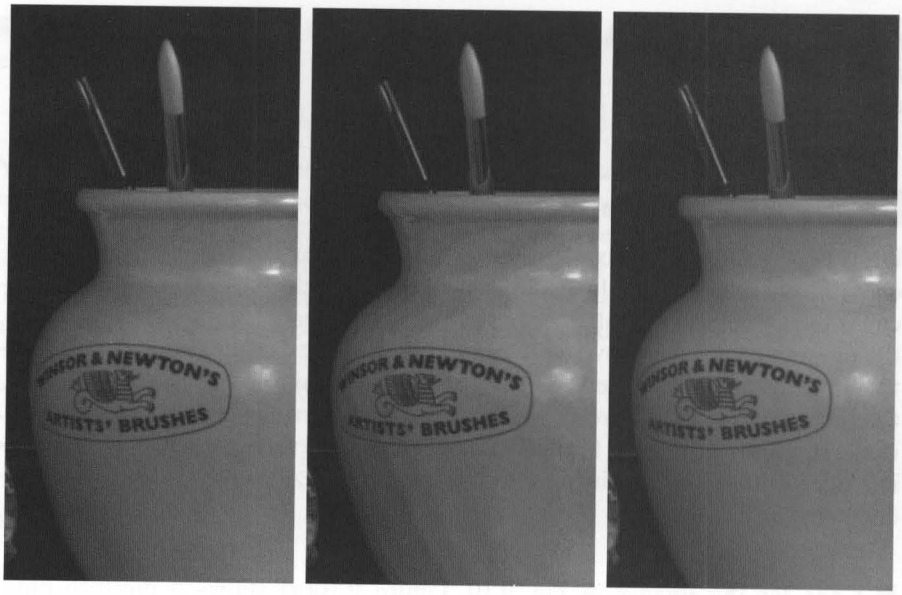
■ Consider the images in Fig. 9.10. Figure 9.10(a) shows a monochrome image with 256 gray levels. Figure 9.10(b) is the same image after uniform quantization to four bits or 16 possible levels. The resulting compression ratio is 2 : 1. Note that false contouring is present in the previously smooth regions of the original image. This is the natural visual effect of more coarsely representing the gray levels of the image.

Figure 9.10(c) illustrates the significant improvements possible with quantization that takes advantage of the peculiarities of the human visual system. Although the compression resulting from this second quantization also is 2 : 1, false contouring is greatly reduced at the expense of some additional but less objectionable graininess. Note that in either case, decompression is both unnecessary and impossible (i.e., quantization is an irreversible operation). ■

The method used to produce Fig. 9.10(c) is called *improved gray-scale* (IGS) *quantization*. It recognizes the eye's inherent sensitivity to edges and breaks them up by adding to each pixel a pseudorandom number, which is generated from the low-order bits of neighboring pixels, before quantizing the result. Because the low order bits are fairly random, this amounts to adding a level of randomness (that depends on the local characteristics of the image) to the artificial edges normally associated with false contouring. Function `quantize`, listed next, performs both IGS quantization and the traditional low-order bit truncation. Note that the IGS implementation is vectorized so that input `x` is processed one column at a time. To generate a column of the 4-bit result in Fig. 9.10(c), a column sum `s`—initially set to all zeros—is formed as the sum of one column of `x` and the four least significant bits of the existing (previously generated) sums. If the four most significant bits of any `x` value are 1111_2 however, 0000_2 is added instead. The four most significant bits of the resulting sums are then used as the coded pixel values for the column being processed.

a b c

FIGURE 9.10
(a) Original image. (b) Uniform quantization to 16 levels. (c) IGS quantization to 16 levels.



quantize

```
function y = quantize(x, b, type)
%QUANTIZE Quantizes the elements of a UINT8 matrix.
%   Y = QUANTIZE(X, B, TYPE) quantizes X to B bits. Truncation is
%   used unless TYPE is 'igs' for Improved Gray Scale quantization.

error(nargchk(2, 3, nargin));          % Check input arguments
if ndims(x) ~= 2 || ~isreal(x) || ...
     ~isnumeric(x) || ~isa(x, 'uint8')
   error('The input must be a UINT8 numeric matrix.');
end

% Create bit masks for the quantization
lo = uint8(2 ^ (8 - b) - 1);
hi = uint8(2 ^ 8 - double(lo) - 1);

% Perform standard quantization unless IGS is specified
if nargin < 3 || ~strcmpi(type, 'igs')
   y = bitand(x, hi);

% Else IGS quantization. Process column-wise. If the MSB's of the
% pixel are all 1's, the sum is set to the pixel value. Else, add
% the pixel value to the LSB's of the previous sum. Then take the
% MSB's of the sum as the quantized value.
else
   [m, n] = size(x);                              s = zeros(m, 1);
   hitest = double(bitand(x, hi) ~= hi);          x = double(x);
   for j = 1:n
      s = x(:, j) + hitest(:, j) .* double(bitand(uint8(s), lo));
```

strcmpi

To compare string `s1` and `s2` ignoring case, use `strcmpi(s1, s2)`.

```
      y(:, j) = bitand(uint8(s), hi);
   end
end
```

Improved gray-scale quantization is typical of a large group of quantization procedures that operate directly on the gray levels of the image to be compressed. They usually entail a decrease in the image's spatial and/or gray-scale resolution. If the image is first mapped to reduce interpixel redundancies, however, the quantization can lead to other types of image degradation—like blurred edges (high-frequency detail loss) when a 2-D frequency transform is used to decorrelate the data.

EXAMPLE 9.7: Combining IGS quantization with lossless predictive and Huffman coding.

■ Although the quantization used to produce Fig. 9.10(c) removes a great deal of psychovisual redundancy with little impact on perceived image quality, further compression can be achieved by employing the techniques of the previous two sections to reduce the resulting image's interpixel and coding redundancies. In fact, we can more than double the 2 : 1 compression of IGS quantization alone. The following sequence of commands combines IGS quantization, lossless predictive coding, and Huffman coding to compress the image of Fig. 9.10(a) to less than a quarter of its original size:

```
>> f = imread('Brushes.tif');
>> q = quantize(f, 4, 'igs');
>> qs = double(q) / 16;
>> e = mat2lpc(qs);
>> c = mat2huff(e);
>> imratio(f, c)
ans =
    4.1420
```

Encoded result c can be decompressed by the inverse sequence of operations (without 'inverse quantization'):

```
>> ne = huff2mat(c);
>> nqs = lpc2mat(ne);
>> nq = 16 * nqs;
>> compare(q, nq)
ans =
    0
>> compare(f, nq)
ans =
    6.8382
```

Note that the root-mean-square error of the decompressed image is about 7 gray levels—and that this error results from the quantization step alone. ■

9.5 JPEG Compression

The techniques of the previous sections operate directly on the pixels of an image and thus are *spatial domain methods*. In this section, we consider a family of popular compression standards that are based on modifying the transform of an image. Our objectives are to introduce the use of 2-D transforms in image compression, to provide additional examples of how to reduce the image redundancies discussed in Section 9.2 through 9.4, and to give the reader a feel for the state of the art in image compression. The standards presented (although we consider only approximations of them) are designed to handle a wide range of image types and compression requirements.

In *transform coding*, a reversible, linear transform like the DFT of Chapter 4 or the *discrete cosine transform* (DCT)

$$T(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\alpha(u)\alpha(v)\cos\left[\frac{(2x+1)u\pi}{2M}\right]\cos\left[\frac{(2y+1)v\pi}{2N}\right]$$

where

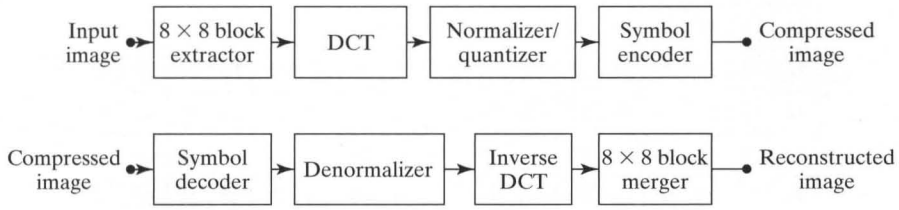
$$\alpha(u) = \begin{cases} \sqrt{\dfrac{1}{M}} & u = 0 \\ \sqrt{\dfrac{2}{M}} & u = 1, 2, \ldots, M-1 \end{cases}$$

[and similarly for $\alpha(v)$] is used to map an image into a set of transform coefficients, which are then quantized and coded. For most natural images, a significant number of the coefficients have small magnitudes and can be coarsely quantized (or discarded entirely) with little image distortion.

9.5.1 JPEG

One of the most popular and comprehensive continuous tone, still frame compression standards is the JPEG (for Joint Photographic Experts Group) standard. In the JPEG *baseline coding standard*, which is based on the discrete cosine transform and is adequate for most compression applications, the input and output images are limited to 8 bits, while the quantized DCT coefficient values are restricted to 11 bits. As can be seen in the simplified block diagram of Fig. 9.11(a), the compression itself is performed in four sequential steps: 8×8 subimage extraction, DCT computation, quantization, and variable-length code assignment.

The first step in the JPEG compression process is to subdivide the input image into nonoverlapping pixel blocks of size 8×8. They are subsequently processed left to right, top to bottom. As each 8×8 block or subimage is processed, its 64 pixels are level shifted by subtracting 2^{m-1} where 2^m is the number of gray levels in the image, and its 2-D discrete cosine transform is computed. The resulting coefficients are then simultaneously denormalized and quantized in accordance with



a
b

FIGURE 9.11
JPEG block diagram: (a) encoder and (b) decoder.

$$\hat{T}(u,v) = \text{round}\left[\frac{T(u,v)}{Z(u,v)}\right]$$

where $\hat{T}(u,v)$ for $u,v = 0, 1, \ldots, 7$ are the resulting denormalized and quantized coefficients, $T(u,v)$ is the DCT of an 8×8 block of image $f(x,y)$, and $Z(u,v)$ is a *transform normalization array* like that of Fig. 9.12(a). By scaling $Z(u,v)$, a variety of compression rates and reconstructed image qualities can be achieved.

After each block's DCT coefficients are quantized, the elements of $\hat{T}(u,v)$ are reordered in accordance with the zigzag pattern of Fig. 9.12(b). Since the resulting one-dimensionally reordered array (of quantized coefficients) is qualitatively arranged according to increasing spatial frequency, the symbol coder of Fig. 9.11(a) is designed to take advantage of the long runs of zeros that normally result from the reordering. In particular, the nonzero AC coefficients [i.e., all $\hat{T}(u,v)$ except $u = v = 0$] are coded using a variable-length code that defines the coefficient's value *and* number of preceding zeros. The DC coefficient [i.e., $\hat{T}(0,0)$] is difference coded relative to the DC coefficient of the previous subimage. Default AC and DC Huffman coding tables are provided by the standard, but the user is free to construct custom tables, as well as normalization arrays, which may in fact be adapted to the characteristics of the image being compressed.

While a full implementation of the JPEG standard is beyond the scope of this chapter, the following M-file approximates the baseline coding process:

im2jpeg

```
function y = im2jpeg(x, quality, bits)
%IM2JPEG Compresses an image using a JPEG approximation.
%   Y = IM2JPEG(X, QUALITY) compresses image X based on 8 x 8 DCT
%   transforms, coefficient quantization, and Huffman symbol
%   coding. Input BITS is the bits/pixel used to for unsigned
%   integer input; QUALITY determines the amount of information that
%   is lost and compression achieved.  Y is an encoding structure
%   containing fields:
%
%      Y.size      Size of X
%      Y.bits      Bits/pixel of X
%      Y.numblocks Number of 8-by-8 encoded blocks
%      Y.quality   Quality factor (as percent)
%      Y.huffman   Huffman encoding structure, as returned by
%                  MAT2HUFF
%
```

a b

FIGURE 9.12
(a) The JPEG default normalization array. (b) The JPEG zigzag coefficient ordering sequence.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

```
%   See also JPEG2IM.

error(nargchk(1, 3, nargin));              % Check input arguments
if ndims(x) ~= 2 || ~isreal(x) || ~isnumeric(x) || ~isinteger(x)
   error('The input image must be unsigned integer.');
end
if nargin < 3
   bits = 8;      % Default value for quality.
end
if bits < 0 || bits > 16
    error('The input image must have 1 to 16 bits/pixel.');
end
if nargin < 2
   quality = 1;   % Default value for quality.
end
if quality <= 0
   error('Input parameter QUALITY must be greater than zero.');
end

m = [16 11  10  16  24  40  51  61        % JPEG normalizing array
     12  12  14  19  26  58  60  55       % and zig-zag redordering
     14  13  16  24  40  57  69  56       % pattern.
     14  17  22  29  51  87  80  62
     18  22  37  56  68  109 103 77
     24  35  55  64  81  104 113 92
     49  64  78  87  103 121 120 101
     72  92  95  98  112 100 103 99] * quality;

order = [1 9  2  3  10 17 25 18 11 4  5  12 19 26 33  ...
         41 34 27 20 13 6  7  14 21 28 35 42 49 57 50  ...
         43 36 29 22 15 8  16 23 30 37 44 51 58 59 52  ...
         45 38 31 24 32 39 46 53 60 61 54 47 40 48 55  ...
         62 63 56 64];

[xm, xn] = size(x);                        % Get input size.
x = double(x) - 2^(round(bits) - 1);       % Level shift input
t = dctmtx(8);                             % Compute 8 x 8 DCT matrix

% Compute DCTs of 8x8 blocks and quantize the coefficients.
```

```
y = blkproc(x, [8 8], 'P1 * x * P2', t, t');
y = blkproc(y, [8 8], 'round(x ./ P1)', m);

y = im2col(y, [8 8], 'distinct');  % Break 8x8 blocks into columns
xb = size(y, 2);                   % Get number of blocks
y = y(order, :);                   % Reorder column elements

eob = max(y(:)) + 1;               % Create end-of-block symbol
r = zeros(numel(y) + size(y, 2), 1);
count = 0;
for j = 1:xb                       % Process 1 block (col) at a time
   i = find(y(:, j), 1, 'last' );  % Find last non-zero element
   if isempty(i)                   % No nonzero block values
      i = 0;
   end
   p = count + 1;
   q = p + i;
   r(p:q) = [y(1:i, j); eob];      % Truncate trailing 0's, add EOB,
   count = count + i + 1;          % and add to output vector
end

r((count + 1):end) = [];           % Delete unusued portion of r

y           = struct;
y.size      = uint16([xm xn]);
y.bits      = uint16(bits);
y.numblocks = uint16(xb);
y.quality   = uint16(quality * 100);
y.huffman   = mat2huff(r);
```

In accordance with the block diagram of Fig. 9.11(a), function `im2jpeg` processes distinct 8×8 sections or *blocks* of input image `x` one block at a time (rather than the entire image at once). Two specialized block processing functions—`blkproc` and `im2col`—are used to simplify the computations. Function `blkproc`, whose standard syntax is

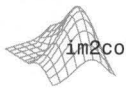
```
B = blkproc(A, [M N], FUN, P1, P2, ...)
```



streamlines or automates the entire process of dealing with images in blocks. It accepts an input image `A`, along with the size (`[M N]`) of the blocks to be processed, a function (`FUN`) to use in processing them, and some number of optional input parameters `P1, P2, ...` for block processing function `FUN`. Function `blkproc` then breaks `A` into `M` x `N` blocks (including any zero padding that may be necessary), calls function `FUN` with each block and parameters `P1, P2, ...`, and reassembles the results into output image `B`.

The second specialized block processing function used by `im2jpeg` is function `im2col`. When `blkproc` is not appropriate for implementing a specific block-oriented operation, `im2col` can often be used to rearrange the input so that the operation can be coded in a simpler and more efficient manner (e.g., by allowing the operation to be vectorized). The output of `im2col` is a matrix

in which each column contains the elements of one distinct block of the input image. Its standardized format is




```
B = im2col(A, [M N], 'distinct')
```

where parameters `A`, `B`, and `[M N]` are as were defined previously for function `blkproc`. String `'distinct'` tells `im2col` that the blocks to be processed are nonoverlapping; alternative string `'sliding'` signals the creation of one column in `B` for every pixel in `A` (as though a block were slid across the image).

In `im2jpeg`, function `blkproc` is used to facilitate both DCT computation and coefficient denormalization and quantization, while `im2col` is used to simplify the quantized coefficient reordering and zero run detection. Unlike the JPEG standard, `im2jpeg` detects only the final run of zeros in each reordered coefficient block, replacing the entire run with the single `eob` symbol. Finally, we note that although MATLAB provides an efficient FFT-based function for large image DCTs (refer to MATLAB's help for function `dct2`), `im2jpeg` uses an alternate matrix formulation:

$$\mathbf{T} = \mathbf{HFH}^T$$

where $\mathbf{F}$ is an 8×8 block of image $f(x, y)$, $\mathbf{H}$ is an 8×8 DCT transformation matrix generated by `dctmtx(8)`, and $\mathbf{T}$ is the resulting DCT of $\mathbf{F}$. Note that the ^{T} is used to denote the transpose operation. In the absence of quantization, the inverse DCT of $\mathbf{T}$ is

$$\mathbf{F} = \mathbf{H}^T\mathbf{TH}$$

This formulation is particularly effective when transforming small square images (like JPEG's 8×8 DCTs). Thus, the statement

```
y = blkproc(x, [8 8], 'P1 * x * P2', h, h')
```

computes the DCTs of image `x` in 8×8 blocks, using DCT transform matrix `h` and transpose `h'` as parameters `P1` and `P2` of the DCT matrix multiplication, `P1 * x * P2`.

Similar block processing and matrix-based transformations [see Fig. 9.11(b)] are required to decompress an `im2jpeg` compressed image. Function `jpeg2im`, listed next, performs the necessary sequence of inverse operations (with the obvious exception of quantization). It uses generic function




```
A = col2im(B,[M N],[MM NN],'distinct')
```

to re-create a 2-D image from the columns of matrix `z`, where each 64-element column is an 8×8 block of the reconstructed image. Parameters `A, B, [M N],` and `'distinct'` are as defined for function `im2col`, while array `[MM NN]` specifies the dimensions of output image `A`.

jpeg2im

```
function x = jpeg2im(y)
%JPEG2IM Decodes an IM2JPEG compressed image.
%   X = JPEG2IM(Y) decodes compressed image Y, generating
%   reconstructed approximation X.  Y is a structure generated by
%   IM2JPEG.
%
%   See also IM2JPEG.
error(nargchk(1, 1, nargin));              % Check input arguments

m = [16 11  10  16  24  40  51  61         % JPEG normalizing array
     12  12  14  19  26  58  60  55        % and zig-zag reordering
     14  13  16  24  40  57  69  56        % pattern.
     14  17  22  29  51  87  80  62
     18  22  37  56  68  109 103 77
     24  35  55  64  81  104 113 92
     49  64  78  87  103 121 120 101
     72  92  95  98  112 100 103 99];

order = [1 9  2  3  10 17 25 18 11 4  5  12 19 26 33  ...
         41 34 27 20 13 6  7  14 21 28 35 42 49 57 50  ...
         43 36 29 22 15 8  16 23 30 37 44 51 58 59 52  ...
         45 38 31 24 32 39 46 53 60 61 54 47 40 48 55  ...
         62 63 56 64];
rev = order;                               % Compute inverse ordering
for k = 1:length(order)
   rev(k) = find(order == k);
end

m = double(y.quality) / 100 * m;           % Get encoding quality.
xb = double(y.numblocks);                  % Get x blocks.
sz = double(y.size);
xn = sz(2);                                % Get x columns.
xm = sz(1);                                % Get x rows.
x = huff2mat(y.huffman);                   % Huffman decode.
eob = max(x(:));                           % Get end-of-block symbol

z = zeros(64, xb);   k = 1;                % Form block columns by copying
for j = 1:xb                               % successive values from x into
   for i = 1:64                            % columns of z, while changing
      if x(k) == eob                       % to the next column whenever
         k = k + 1;   break;               % an EOB symbol is found.
      else
         z(i, j) = x(k);
         k = k + 1;
      end
   end
end

z = z(rev, :);                                  % Restore order
x = col2im(z, [8 8], [xm xn], 'distinct');      % Form matrix blocks
x = blkproc(x, [8 8], 'x .* P1', m);            % Denormalize DCT
t = dctmtx(8);                                  % Get 8 x 8 DCT matrix
x = blkproc(x, [8 8], 'P1 * x * P2', t', t);    % Compute block DCT-1
```

```
x = x + double(2^(y.bits - 1));                % Level shift
if y.bits <= 8
    x = uint8(x);
else
    x = uint16(x);
end
```

EXAMPLE 9.8: JPEG compression.

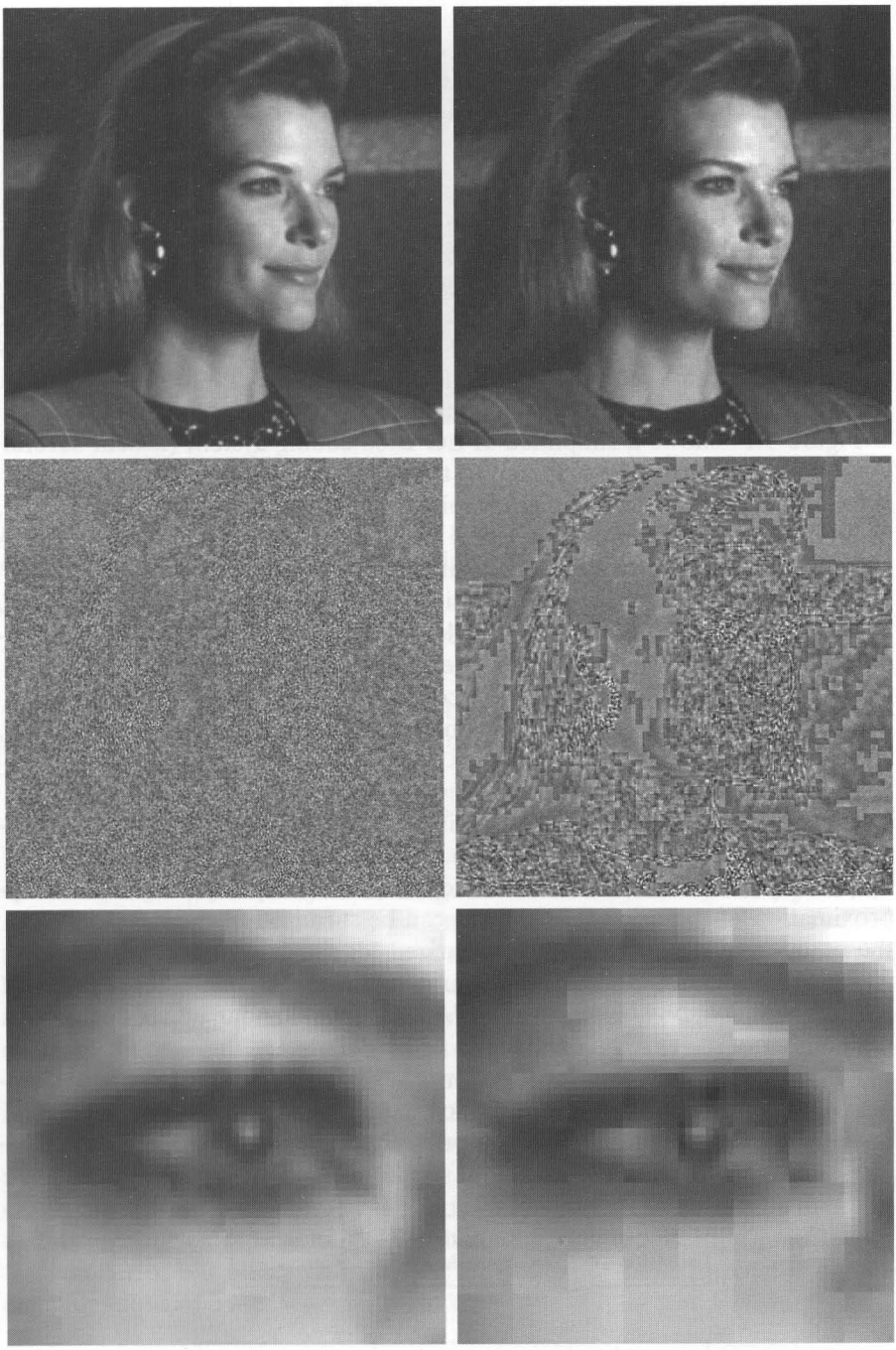
■ Figures 9.13(a) and (b) show two JPEG coded and subsequently decoded approximations of the monochrome image in Fig. 9.4(a). The first result, which provides a compression ratio of about 18 to 1, was obtained by direct application of the normalization array in Fig. 9.12(a). The second, which compresses the original image by a ratio of 42 to 1, was generated by multiplying (scaling) the normalization array by 4.

The differences between the original image of Fig. 9.4(a) and the reconstructed images of Figs. 9.13(a) and (b) are shown in Figs. 9.13(c) and (d) respectively. Both images have been scaled to make the errors more visible. The corresponding rms errors are 2.4 and 4.4 gray levels. The impact of these errors on picture quality is more visible in the zoomed images of Figs. 9.13(e) and (f). These images show a magnified section of Figs. 9.13(a) and (b), respectively, and allow a better assessment of the subtle differences between the reconstructed images. [Figure 9.4(b) shows the zoomed original.] Note the *blocking artifact* that is present in both zoomed approximations.

The images in Fig. 9.13 and the numerical results just discussed were generated with the following sequence of commands:

```
>> f = imread('Tracy.tif');
>> c1 = im2jpeg(f);
>> f1 = jpeg2im(c1);
>> imratio(f, c1)
ans =
    18.4090
>> compare(f, f1, 3)
ans =
    2.4329
>> c4 = im2jpeg(f, 4);
>> f4 = jpeg2im(c4);
>> imratio(f, c4)
ans =
    43.3153
>> compare(f, f4, 3)
ans =
    4.4053
```

These results differ from those that would be obtained in a real JPEG baseline coding environment because `im2jpeg` approximates the JPEG standard's Huffman encoding process. Two principal differences are noteworthy: (1) In



a b
c d
e f

FIGURE 9.13
Left column: Approximations of Fig. 9.4 using the DCT and normalization array of Fig. 9.12(a). Right column: Similar results with the normalization array scaled by a factor of 4.

the standard, all runs of coefficient zeros are Huffman coded, while `im2jpeg` only encodes the terminating run of each block; and (2) the encoder and decoder of the standard are based on a known (default) Huffman code, while `im2jpeg` carries the information needed to reconstruct the encoding Huffman code words on an image to image basis. Using the standard, the compressions ratios noted above would be approximately doubled. ■

9.5.2 JPEG 2000

Like the initial JPEG release of the previous section, JPEG 2000 is based on the idea that the coefficients of a transform that decorrelates the pixels of an image can be coded more efficiently than the original pixels themselves. If the transform's basis functions—wavelets in the JPEG 2000 case—pack most of the important visual information into a small number of coefficients, the remaining coefficients can be quantized coarsely or truncated to zero with little image distortion.

Figure 9.14 shows a simplified JPEG 2000 coding system (absent several optional operations). The first step of the encoding process, as in the original JPEG standard, is to level shift the pixels of the image by subtracting 2^{m-1}, where 2^m is the number of gray levels in the image. The one-dimensional discrete wavelet transform of the rows and the columns of the image can then be computed. For error-free compression, the transform used is biorthogonal, with a 5-3 coefficient scaling and wavelet vector. In lossy applications, a 9-7 coefficient scaling-wavelet vector (see the `wavefilter` function of Chapter 8) is employed. In either case, the initial decomposition results in four subbands—a low-resolution approximation of the image and the image's horizontal, vertical, and diagonal frequency characteristics.

Repeating the decomposition process N_L times, with subsequent iterations restricted to the previous decomposition's approximation coefficients, produces an N_L-scale wavelet transform. Adjacent scales are related spatially by powers of 2, and the lowest scale contains the only explicitly defined approximation of the original image. As can be surmised from Fig. 9.15, where the notation of the standard is summarized for the case of $N_L = 2$, a general N_L-scale transform contains $3N_L + 1$ subbands whose coefficients are denoted a_b for $b = N_L LL, N_L HL, \ldots, 1HL, 1LH, 1HH$. The standard does not specify the number of scales to be computed.

After the N_L-scale wavelet transform has been computed, the total number of transform coefficients is equal to the number of samples in the original image—but the important visual information is concentrated in a few coef-

a
b

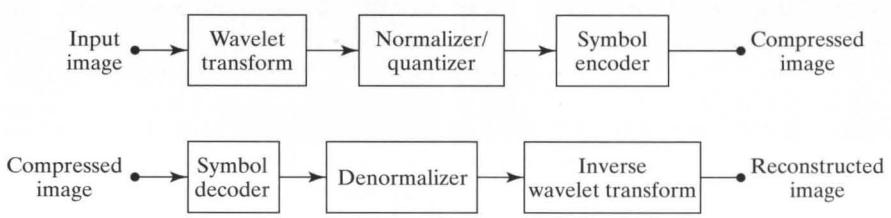


FIGURE 9.14 JPEG 2000 block diagram: (a) encoder and (b) decoder.

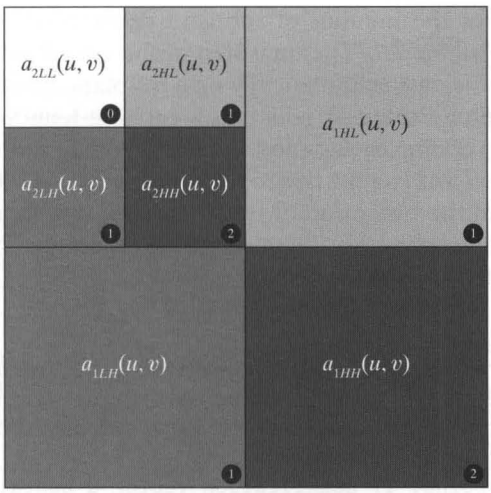


FIGURE 9.15
JPEG 2000 two-scale wavelet transform coefficient notation and analysis gain (in the circles).

ficients. To reduce the number of bits needed to represent them, coefficient $a_b(u,v)$ of subband b is quantized to value $q_b(u,v)$ using

$$q_b(u,v) = \text{sign}[a_b(u,v)] \cdot \text{floor}\left[\frac{|a_b(u,v)|}{\Delta_b}\right]$$

where the "sign" and "floor" operators behave like MATLAB functions of the same name (i.e., functions `sign` and `floor`). Quantization step size Δ_b is

$$\Delta_b = 2^{R_b - \varepsilon_b}\left(1 + \frac{\mu_b}{2^{11}}\right)$$

where R_b is the nominal dynamic range of subband b, and ε_b and μ_b are the number of bits allotted to the exponent and mantissa of the subband's coefficients. The nominal dynamic range of subband b is the sum of the number of bits used to represent the original image and the analysis gain bits for subband b. Subband analysis gain bits follow the simple pattern shown in Fig. 9.15. For example, there are two analysis gain bits for subband $b = 1HH$.

For error-free compression, $\mu_b = 0$ and $R_b = \varepsilon_b$ so that $\Delta_b = 1$. For irreversible compression, no particular quantization step size is specified. Instead, the number of exponent and mantissa bits must be provided to the decoder on a subband basis, called *explicit quantization*, or for the $N_L LL$ subband only, called *implicit quantization*. In the latter case, the remaining subbands are quantized using extrapolated $N_L LL$ subband parameters. Letting ε_0 and μ_0 be the number of bits allocated to the $N_L LL$ subband, the extrapolated parameters for subband b are

$$\mu_b = \mu_0$$
$$\varepsilon_b = \varepsilon_0 + nsd_b - nsd_0$$

where nsd_b denotes the number of subband decomposition levels from the original image to subband b. The final step of the encoding process is to code the quantized coefficients arithmetically on a bit-plane basis. Although not discussed in the chapter, *arithmetic coding* is a variable-length coding procedure that, like Huffman coding, is designed to reduce coding redundancy.

Custom function `im2jpeg2k` approximates the JPEG 2000 coding process of Fig. 9.14(a) with the exception of the arithmetic symbol coding. As can be seen in the following listing, Huffman encoding augmented by zero run-length coding is substituted for simplicity.

`im2jpeg2k`

```
function y = im2jpeg2k(x, n, q)
%IM2JPEG2K Compresses an image using a JPEG 2000 approximation.
%   Y = IM2JPEG2K(X, N, Q) compresses image X using an N-scale JPEG
%   2K wavelet transform, implicit or explicit coefficient
%   quantization, and Huffman symbol coding augmented by zero
%   run-length coding. If quantization vector Q contains two
%   elements, they are assumed to be implicit quantization
%   parameters; else, it is assumed to contain explicit subband step
%   sizes.  Y is an encoding structure containing Huffman-encoded
%   data and additional parameters needed by JPEG2K2IM for decoding.
%
%   See also JPEG2K2IM.

global RUNS

error(nargchk(3, 3, nargin));         % Check input arguments

if ndims(x) ~= 2 || ~isreal(x) || ~isnumeric(x) || ~isa(x, 'uint8')
   error('The input must be a UINT8 image.');
end

if length(q) ~= 2 && length(q) ~= 3 * n + 1
   error('The quantization step size vector is bad.');
end

% Level shift the input and compute its wavelet transform
x = double(x) - 128;
[c, s] = wavefast(x, n, 'jpeg9.7');

% Quantize the wavelet coefficients.
q = stepsize(n, q);
sgn = sign(c);     sgn(find(sgn == 0)) = 1;     c = abs(c);
for k = 1:n
   qi = 3 * k - 2;
   c = wavepaste('h', c, s, k, wavecopy('h', c, s, k) / q(qi));
   c = wavepaste('v', c, s, k, wavecopy('v', c, s, k) / q(qi + 1));
   c = wavepaste('d', c, s, k, wavecopy('d', c, s, k) / q(qi + 2));
end
c = wavepaste('a', c, s, k, wavecopy('a', c, s, k) / q(qi + 3));
c = floor(c);       c = c .* sgn;

% Run-length code zero runs of more than 10. Begin by creating
% a special code for 0 runs ('zrc') and end-of-code ('eoc') and
```

```
% making a run-length table.
zrc = min(c(:)) - 1;      eoc = zrc - 1;      RUNS = 65535;

% Find the run transition points: 'plus' contains the index of the
% start of a zero run; the corresponding 'minus' is its end + 1.
z = c == 0;                  z = z - [0 z(1:end - 1)];
plus = find(z == 1);         minus = find(z == -1);

% Remove any terminating zero run from 'c'.
if length(plus) ~= length(minus)
   c(plus(end):end) = [];      c = [c eoc];
end

% Remove all other zero runs (based on 'plus' and 'minus') from 'c'.
for i = length(minus):-1:1
   run = minus(i) - plus(i);
   if run > 10
      ovrflo = floor(run / 65535);    run = run - ovrflo * 65535;
      c = [c(1:plus(i) - 1) repmat([zrc 1], 1, ovrflo) zrc ...
           runcode(run) c(minus(i):end)];
   end
end

% Huffman encode and add misc. information for decoding.
y.runs    = uint16(RUNS);
y.s       = uint16(s(:));
y.zrc     = uint16(-zrc);
y.q       = uint16(100 * q');
y.n       = uint16(n);
y.huffman = mat2huff(c);

%-------------------------------------------------------------------%
function y = runcode(x)
% Find a zero run in the run-length table. If not found, create a
% new entry in the table. Return the index of the run.

global RUNS
y = find(RUNS == x);
if length(y) ~= 1
   RUNS = [RUNS; x];
   y = length(RUNS);
end

%-------------------------------------------------------------------%
function q = stepsize(n, p)
% Create a subband quantization array of step sizes ordered by
% decomposition (first to last) and subband (horizontal, vertical,
% diagonal, and for final decomposition the approximation subband).

if length(p) == 2              % Implicit Quantization
   q = [];
   qn = 2 ^ (8 - p(2) + n) * (1 + p(1) / 2 ^ 11);
   for k = 1:n
      qk = 2 ^ -k * qn;
```

```
      q = [q (2 * qk) (2 * qk) (4 * qk)];
   end
   q = [q qk];
else                                % Explicit Quantization
   q = p;
end

q = round(q * 100) / 100;           % Round to 1/100th place
if any(100 * q > 65535)
   error('The quantizing steps are not UINT16 representable.');
end
if any(q == 0)
   error('A quantizing step of 0 is not allowed.');
end
```

JPEG 2000 decoders simply invert the operations described previously. After decoding the arithmetically coded coefficients, a user-selected number of the original image's subbands are reconstructed. Although the encoder may have arithmetically encoded M_b bit-planes for a particular subband, the user—due to the embedded nature of the codestream—may choose to decode only N_b bit-planes. This amounts to quantizing the coefficients using a step size of $2^{M_b - N_b} \bullet \Delta_b$. Any non-decoded bits are set to zero and the resulting coefficients, denoted $\overline{q}_b(u,v)$ are denormalized using

$$R_{q_b}(u,v) = \begin{cases} (\overline{q}_b(u,v) + 2^{M_b - N_b(u,v)}) \bullet \Delta_b & \overline{q}_b(u,v) > 0 \\ (\overline{q}_b(u,v) - 2^{M_b - N_b(u,v)}) \bullet \Delta_b & \overline{q}_b(u,v) < 0 \\ 0 & \overline{q}_b(u,v) = 0 \end{cases}$$

where $R_{q_b}(u,v)$ denotes a denormalized transform coefficient and $N_b(u,v)$ is the number of decoded bit-planes for $\overline{q}_b(u,v)$. The denormalized coefficients are then inverse transformed and level shifted to yield an approximation of the original image. Custom function `jpeg2k2im` approximates this process, reversing the compression of `im2jpeg2k` introduced earlier.

`jpeg2k2im`

```
function x = jpeg2k2im(y)
%JPEG2K2IM Decodes an IM2JPEG2K compressed image.
%   X = JPEG2K2IM(Y) decodes compressed image Y, reconstructing an
%   approximation of the original image X.  Y is an encoding
%   structure returned by IM2JPEG2K.
%
%   See also IM2JPEG2K.

error(nargchk(1, 1, nargin));       % Check input arguments

% Get decoding parameters: scale, quantization vector, run-length
% table size, zero run code, end-of-data code, wavelet bookkeeping
% array, and run-length table.
n = double(y.n);
```

```
q = double(y.q) / 100;
runs = double(y.runs);
zrc = -double(y.zrc);
eoc = zrc - 1;
s = double(y.s);
s = reshape(s, n + 2, 2);

% Compute the size of the wavelet transform.
cl = prod(s(1, :));
for i = 2:n + 1
   cl = cl + 3 * prod(s(i, :));
end

% Perform Huffman decoding followed by zero run decoding.
r = huff2mat(y.huffman);

c = [];   zi = find(r == zrc);    i = 1;
for j = 1:length(zi)
   c = [c r(i:zi(j) - 1) zeros(1, runs(r(zi(j) + 1)))];
   i = zi(j) + 2;
end

zi = find(r == eoc);                % Undo terminating zero run
if length(zi) == 1                  % or last non-zero run.
   c = [c r(i:zi - 1)];
   c = [c zeros(1, cl - length(c))];
else
   c = [c r(i:end)];
end

% Denormalize the coefficients.
c = c + (c > 0) - (c < 0);
for k = 1:n
   qi = 3 * k - 2;
   c = wavepaste('h', c, s, k, wavecopy('h', c, s, k) * q(qi));
   c = wavepaste('v', c, s, k, wavecopy('v', c, s, k) * q(qi + 1));
   c = wavepaste('d', c, s, k, wavecopy('d', c, s, k) * q(qi + 2));
end
c = wavepaste('a', c, s, k, wavecopy('a', c, s, k) * q(qi + 3));

% Compute the inverse wavelet transform and level shift.
x = waveback(c, s, 'jpeg9.7', n);
x = uint8(x + 128);
```

The principal difference between the wavelet-based JPEG 2000 system of Fig. 9.14 and the DCT-based JPEG system of Fig. 9.11 is the omission of the latter's subimage processing stages. Because wavelet transforms are both computationally efficient and inherently local (i.e., their basis functions are limited in duration), subdivision of the image into blocks is unnecessary. As will be seen in the following example, the removal of the subdivision step eliminates the blocking artifact that characterizes DCT-based approximations at high compression ratios.

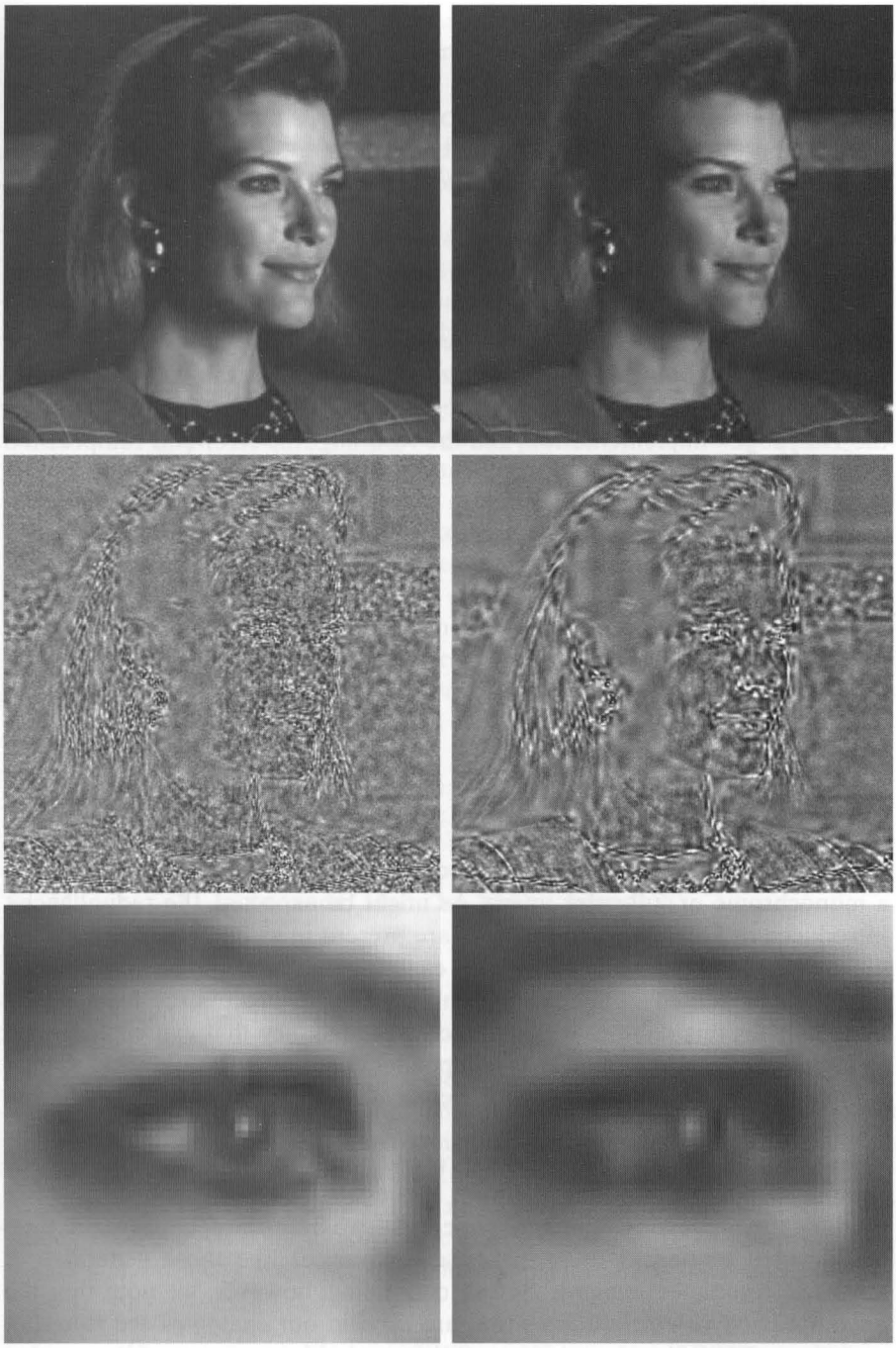
EXAMPLE 9.8: JPEG 2000 compression.

■ Figure 9.16 shows two JPEG 2000 approximations of the monochrome image in Figure 9.4(a). Figure 9.16(a) was reconstructed from an encoding that compressed the original image by 42 : 1. Fig. 9.16(b) was generated from an 88 : 1 encoding. The two results were obtained using a five-scale transform and implicit quantization with $\mu_0 = 8$ and $\varepsilon_0 = 8.5$ and 7, respectively. Because `im2jpeg2k` only approximates the JPEG 2000's bit-plane–oriented arithmetic coding, the compression rates just noted differ from those that would be obtained by a true JPEG 2000 encoder. In fact, the actual rates would increase by approximately a factor of 2.

Since the 42 : 1 compression of the results in the left column of Fig. 9.16 is identical to the compression achieved for the images in the right column of Fig. 9.13 (Example 9.8), Figs. 9.16(a), (c), and (e) can be compared—both qualitatively and quantitatively—to the transform-based JPEG results of Figs. 9.13(b), (d), and (f). A visual comparison reveals a noticeable decrease of error in the wavelet-based JPEG 2000 images. In fact, the rms error of the JPEG 2000–based result in Fig. 9.16(a) is 3.6 gray levels, as opposed to 4.4 gray levels for the corresponding transform-based JPEG result in Fig. 9.13(b). Besides decreasing reconstruction error, JPEG 2000–based coding dramatically increased (in a subjective sense) image quality. This is particularly evident in Fig. 9.16(e). Note that the blocking artifact that dominated the corresponding transform-based result in Fig. 9.13(f) is no longer present.

When the level of compression increases to 88 : 1 as in Fig. 9.16(b), there is a loss of texture in the woman's clothing and blurring of her eyes. Both effects are visible in Figs. 9.16(b) and (f). The rms error of these reconstructions is about 5.9 gray levels. The results of Fig. 9.16 were generated with the following sequence of commands:

```
>> f = imread('Tracy.tif');
>> c1 = im2jpeg2k(f, 5, [8 8.5]);
>> f1 = jpeg2k2im(c1);
>> rms1 = compare(f, f1)

rms1 =
    3.6931

>> cr1 = imratio(f, c1)

cr1 =
    42.1589

>> c2 =im2jpeg2k(f, 5, [8 7]);
>> f2 = jpeg2k2im(c2);
>> rms2 = compare(f, f2)

rms2 =
    5.9172

>> cr2 = imratio(f, c2)
cr2 =
```



a b
c d
e f

FIGURE 9.16
Left column: JPEG 2000 approximations of Fig. 9.4 using five scales and implicit quantization with $\mu_0 = 8$ and $\varepsilon_0 = 8.5$. Right column: Similar results with $\varepsilon_0 = 7$.

```
    87.7323
```

Note that implicit quantization is used when a two-element vector is supplied as argument 3 of `im2jpeg2k`. If the length of this vector is not 2, the function assumes explicit quantization and $3N_L + 1$ step sizes (where N_L is the number of scales to be computed) must be provided. This is one for each subband of the decomposition; they must be ordered by decomposition level (first, second, third, ...) and by subband type (i.e., the horizontal, vertical, diagonal, and approximation). For example,

```
>> c3 =im2jpeg2k(f, 1, [1 1 1 1]);
```

computes a one-scale transform and employs explicit quantization—all four subbands are quantized using step size $\Delta_1 = 1$. That is, the transform coefficients are rounded to the nearest integer. This is the minimal error case for the `im2jpeg2k` implementation, and the resulting rms error and compression rate are

```
>> f3 = jpeg2k2im(c3);
>> rms3 = compare(f, f3)
rms3 =
    1.1234
>> cr3 = imratio(f, c3)
cr3 =
    1.6350
```

■

9.6 Video Compression

A *video* is a sequence of images, called *video frames*, in which each frame is a monochrome or full-color image. As might be expected, the redundancies introduced in Sections 9.2 though 9.4 are present in most video frames—and the compression methods previously examined, as well as the compression standards presented in Section 9.5, can be used to process the frames independently. In this section, we introduce a redundancy that can be exploited to increase the compression that independent processing would yield. Called *temporal redundancy*, it is due to the correlations between pixels in adjacent frames.

In the material that follows, we present both the fundamentals of video compression and the principal Image Processing Toolbox functions that are used for the processing of image sequences—whether the sequences are time-based video sequences or spatial-based sequences like those generated in magnetic resonance imaging. Before continuing, however, we note that the uncompressed video sequences that are used in our examples are stored in *multiframe TIFF* files. A multiframe TIFF can hold a sequence of images that may be read *one at a time* using the following `imread` syntax

```
imread('filename.tif', idx)
```

where `idx` is the integer index of the frame in the sequence to be read. To write uncompressed frames to a multiframe TIFF file, the corresponding `imwrite` syntax is

```
imwrite(f, 'filename', 'Compression', 'none', ...
                'WriteMode', mode)
```

where `mode` is set to `'overwrite'` when writing the initial frame and to `'append'` when writing all other frames. Note that unlike `imread`, `imwrite` does not provide random access to the frames in a multiframe TIFF; frames must be written in the time order in which they occur.

9.6.1 MATLAB Image Sequences and Movies

There are two standard ways to represent a video in the MATLAB workspace. In the first, which is also the simplest, each frame of video is concatenated along the fourth dimension of a four dimensional array. The resulting array is called a MATLAB *image sequence* and its first two dimensions are the row and column dimensions of the concatenated frames. The third dimension is 1 for monochrone (or indexed) images and 3 for full-color images; the fourth dimension is the number of frames in the image sequence. Thus, the following commands read the first and last frames of the 16-frame multiframe TIFF, `'shuttle.tif'`, and build a simple two-frame $256 \times 480 \times 1 \times 2$ monochrome image sequence `s1`:

```
>> i = imread('shuttle.tif', 1);
>> frames = size(imfinfo('shuttle.tif'), 1);
>> s1 = uint8(zeros([size(i) 1 2]));
>> s1(:,:,:,1) = i;
>> s1(:,:,:,2) = imread('shuttle.tif', frames);
>> size(s1)
ans =
   256   480     1     2
```

An alternate way to represent a video in the MATLAB workspace is to embed successive video frames into a matrix of structures called *movie frames*. Each column in the resulting one-row matrix, which is called a MATLAB *movie*, is a structure that includes both a `cdata` field, which holds one frame of video as a 2- or 3-D matrix of `uint8` values, and a `colormap` field, which contains a standard MATLAB color lookup table (see Section 6.1.2). The following commands convert image sequence `s1` (from above) into MATLAB movie `m1`:

```
>> lut = 0:1/255:1;
>> lut = [lut' lut' lut'];
>> m1(1) = im2frame(s1(:,:,:,1), lut);
```

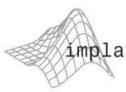


Function `im2frame(x, map)` converts an indexed image `x` and associated colormap `map` into a movie frame. If `x` is full color, `map` is optional and has no effect.

```
>> m1(2) = im2frame(s1(:,:,:,2), lut);
>> size(m1)
ans =
   1     2
>> m1(1)
ans =
       cdata: [256x480 uint8]
    colormap: [256x3 double]
```

As can be seen, movie `m1` is a 1×2 matrix whose elements are structures containing 256×480 `uint8` images and 256×3 lookup tables. Lookup table `lut` is a 1 : 1 grayscale mapping. Finally, note that function `im2frame`, which takes an image and a color lookup table as arguments, is used to build each movie frame.

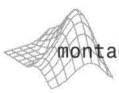
Whether a given video sequence is represented as a standard MATLAB movie or as a MATLAB image sequence, it can be viewed (played, paused, single stepped, etc.) using function `implay`:




```
implay(frms, fps)
```

where `frms` is a MATLAB movie or image sequence and `fps` is an optional frame rate (in frames per second) for the playback. The default frame rate is 20 frames/sec. Figure 9.17 shows the *movie player* that is displayed in response to the `implay(s1)` and/or `implay(m1)` command with `s1` and `m1` as defined above. Note that the *playback toolbar* provides controls that are reminiscent of the controls on a commerical DVD player. In addition, the index of the current frame (the `1` in the `1/2` at the lower right of Fig. 9.17), its type (`I` as opposed to `RGB`), size (`256x480`), as well as the frame rate (`20 fps`) and total number of frames in the movie or image sequence being displayed (the `2` in the `1/2`), is shown along the bottom of the movie player window. Note also that the window can be resized to fit the image being displayed; when the window is smaller than the currently displayed image, scroll bars are added to the sides of the viewing area.

Multiple frames can be simultaneously viewed using the `montage` function:




```
montage(frms, 'Indices', idxes, 'Size', [rows cols])
```

For more information on the parameters that are used in the `montage` function, type `>> help montage`.

Here, `frms` is as defined above, `idxes` is a numeric array that specifies the indices of the frames that are used to populate the montage, and `rows` and `cols` define its shape. Thus, `montage(s1, 'Size', [2 1])` displays a 2×1 montage of the two-frame sequence `s1` (see Fig. 9.18). Recall that `s1` is composed of the first and last frames of `'shuttle.tif'`. As Fig. 9.18 suggests, the biggest visual difference between any frame in `'shuttle.tif'` is the position of the Earth in the background. It moves from left to right with respect to a stationary camera on the shuttle itself.

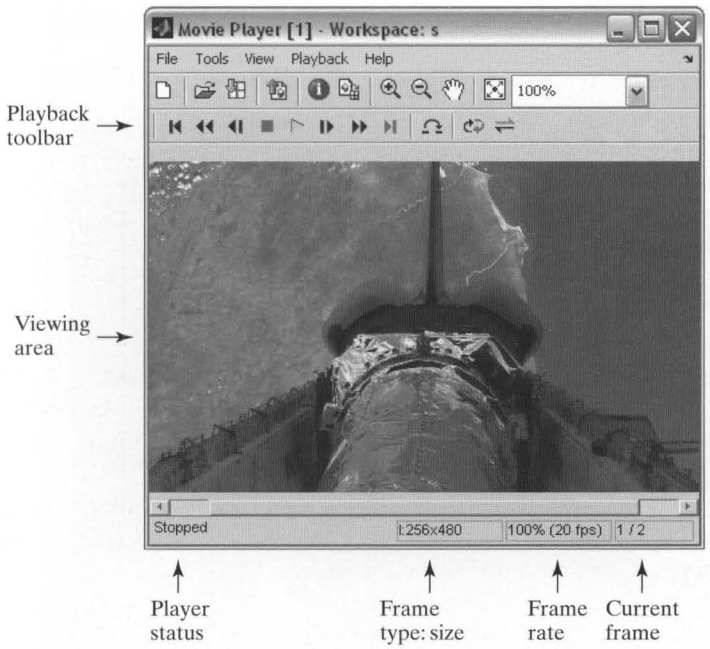


FIGURE 9.17
The toolbox movie player. (Original image courtesy of NASA.)

To conclude the section, we introduce several custom functions that are used for converting between image sequences, movies, and multiframe TIFFs. These functions are included in Appendix C and make it easier to work with multiframe TIFF files. To convert between multiframe TIFFs and MATLAB image sequences, for example, use

```
s = tifs2seq('filename.tif')
```

tifs2seq

and

```
seq2tifs(s, 'filename.tif')
```

seq2tifs

where `s` is a MATLAB image sequence and `'filename.tif'` is a multiframe TIFF file. To perform simlar conversions with MATLAB movies, use

```
m = tifs2movie('filename.tif')
```

tifs2movie

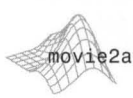
and

```
movie2tifs(m, 'filename.tif')
```

movie2tifs

where `m` is MATLAB movie. Finally, to convert a multiframe TIFF to an *Advanced Video Interleave* (AVI) file for use with the *Windows Media Player*, use `tifs2movie` in conjunction with MATLAB function `movie2avi`:

```
movie2avi(tifs2movie('filename.tif'), 'filename.avi')
```



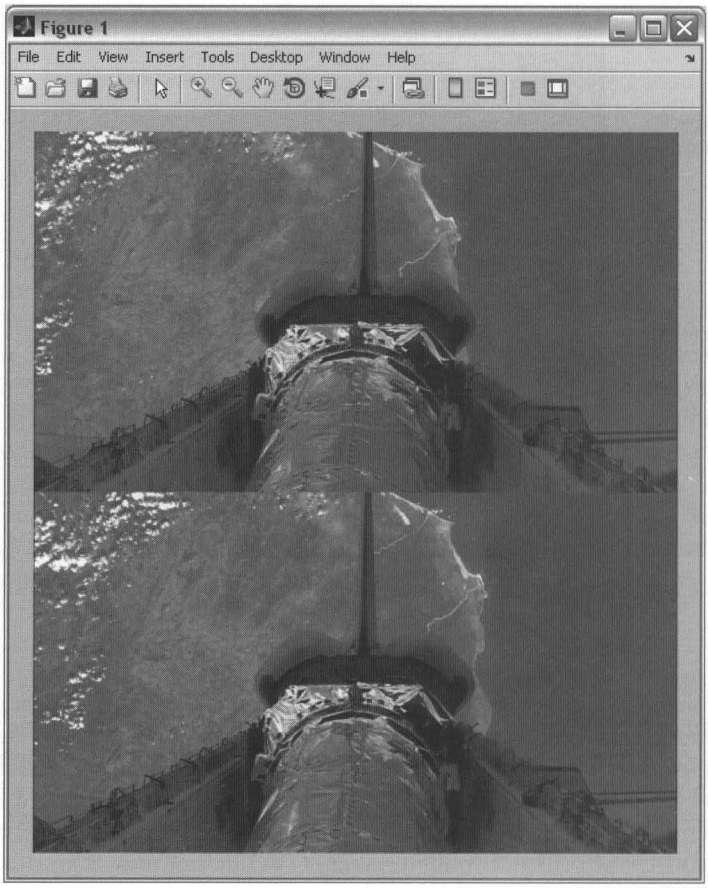


FIGURE 9.18
A montage of two video frames. (Original images courtesy of NASA.)

where `'filename.tif'` is a multiframe TIFF and `'filename.avi'` is the name of the generated AVI file. To view a multiframe TIFF on the toolbox movie player, combine `tifs2movie` with function `implay`:

```
implay(tifs2movie('filename.tif'))
```

9.6.2 Temporal Redundancy and Motion Compensation

Like spatial redundancies, which result from correlations between pixels that are near to one another in space, temporal redundancies are due to correlations between pixels that are close to one another in time. As will be seen in the following example, which parallels Example 9.5 of Section 9.3, both redundancies are addressed in much the same way.

EXAMPLE 9.9: Temporal redundancy.

■ Figure 9.19(a) shows the second frame of the multiframe TIFF whose first and last frames are depicted in Fig. 9.18. As was noted in Sections 9.2 and 9.3, the spatial and coding redundancies that are present in a conventional 8-bit representation of the frame can be removed through the use of Huffman and

linear predictive coding:

```
>> f2 = imread('shuttle.tif', 2);
>> ntrop(f2)
ans =
    6.8440
>> e2 = mat2lpc(f2);
>> ntrop(e2, 512)
ans =
    4.4537
>> c2 = mat2huff(e2);
>> imratio (f2, c2)
ans =
    1.7530
```

Function `mat2lpc` predicts the value of the pixels in `f2` from their immediately preceding neighbors (in space), while `mat2huff` encodes the differences between the predictions and the actual pixel values. The prediction and differencing process results in a compression of 1.753 : 1.

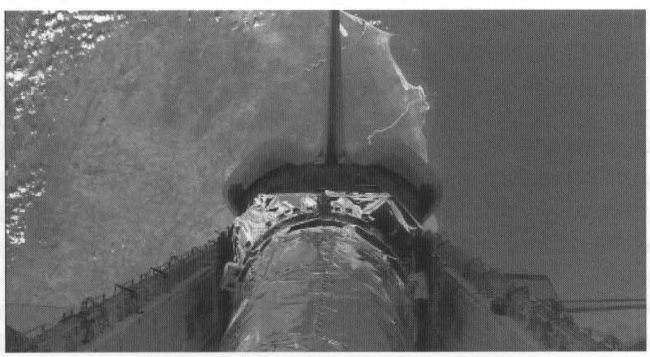
Because `f2` is part of a time sequence of images, we can alternately predict its pixels from the corresponding pixels in the previous frame. Using the first-order linear predictor

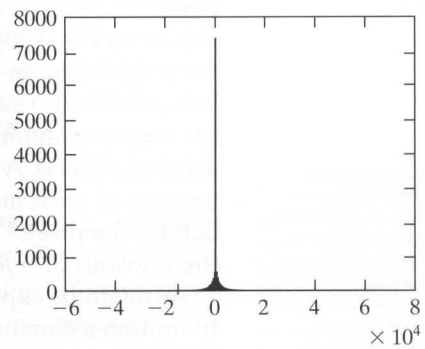
$$\hat{f}(x,y,t) = \text{round}[\alpha f(x,y,t-1)]$$

with $\alpha = 1$ and Huffman encoding the resulting prediction error

$$e(x,y,t) = f(x,y,t-1) - f(x,y,t)$$

we get:





a b

FIGURE 9.19 (a) The second frame of a 16-frame video of the space shuttle in orbit around the Earth. The first and last frames are shown in Fig. 9.18. (b) The histogram of the prediction error resulting from the previous frame prediction in Example 9.9. (Original image courtesy of NASA).

```
>> f1 = imread('shuttle.tif', 1);
>> ne2 = double(f2) - double(f1);
>> ntrop(ne2, 512)
ans =
    3.0267
>> nc2 = mat2huff(ne2);
>> imratio (f2, nc2)
ans =
    2.5756
```

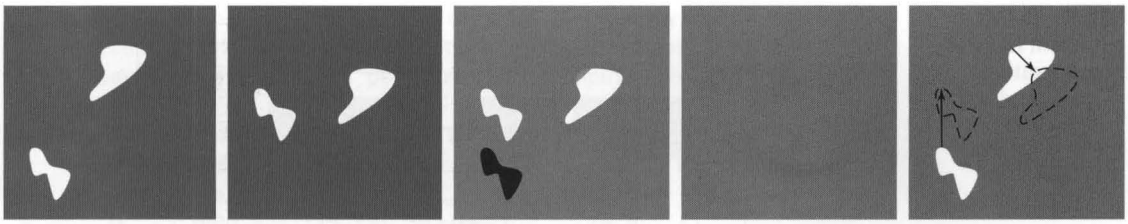
Using an interframe predictor, as opposed to a spatially-oriented previous pixel predictor, the compression is increased to 2.5756. In either case, compression is lossless and due to the fact that the entropy of the resulting prediction residuals (4.4537 bits/pixel for `e2` and 3.0267 bits/pixel for `ne2`), is lower than the entropy of frame `f2`, which is 6.8440 bits/pixel. Note that the histogram of predition residual `ne2` is displayed in Fig. 9.19(b). It is highly peaked around 0 and has a relatively small variance, making it ideal for variable-length Huffman coding. ■

A simple way to increase the accuracy of most interframe predictions is to account for the frame-to-frame motion of objects—a process called *motion compensation*. The basic idea is illustrated in Fig. 9.20, where the (a) and (b) parts of the figure are adjacent frames in a hypothetical video containing two objects in motion. Both objects are white; the background is gray level 75. If the frame shown in Fig. 9.20(b) is encoded using the frame in Fig. 9.20(a) as its predictor (as was done in Example 9.9), the resulting prediction residual contains three values (i.e., −180, 0, and 180). [See Fig. 9.20(c), where the prediction residual is scaled so that gray level 128 corresponds to a prediction error of 0.] If object motion is taken into account, however, the resulting prediction residual will have only one value—0. Note in Fig. 9.20(d) that the motion compensated residual contains no information. Its entropy is 0. Only the *motion vectors* in Fig. 9.20(e) would be needed to reconstruct the frame shown in (b) from the frame in (a). In a non-idealized case, however, both motion vectors and prediction residuals are needed—and the motion vectors are computed for non-overlapping rectangular regions called *macroblocks* rather than individual objects. A single vector then describes the motion (i.e., direction and amount of movement) of every pixel in the associated macroblock; that is, it defines the pixels' horizontal and vertical displacement from their position in the previous or *reference frame*.

The three possible prediction residual values are the differences formed from gray levels 255 (the object white) and 75 (the background gray).

The discussion here assumes that motion vectors are specified to the nearest integer or whole pixel location. If the precision is increased to the sub-pixel (e.g., ½ or ¼ pixel) level, predictions must be interpolated (e.g., using bilinear interpolation) from a combination of pixels in the reference frame.

As might be expected, *motion estimation* is the key to motion compensation. In motion estimation, the motion of each macroblock is measured and encoded into a motion vector. The vector is selected to minimize the error between the associated macroblock pixels and the prediction pixels in the reference frame. One of the most commonly used error measures is the *sum of absolute distortion* (*SAD*)



a b c d e

FIGURE 9.20 (a) and (b) Two frames of a hypothetical video. (c) The scaled prediction residual without motion compensation. (d) The prediction residual after motion compensation. (e) Motion vectors describing the movement of objects.

$$SAD(x,y) = \sum_{i=1}^{m}\sum_{j=1}^{n}|f(x+i, y+j) - p(x+i+dx, y+j+dy)|$$

where x and y are the coordinates of the upper-left pixel of the $m \times n$ macroblock being coded, dx and dy are displacements from its reference frame position, and p is an array of predicted macroblock pixel values. Typically, dx and dy must fall within a limited search region around each macroblock. Values from ± 8 to ± 64 pixels are common, and the horizontal search area is often slightly larger than the vertical area. Given a criterion like SAD, motion estimation is performed by searching for the dx and dy that minimizes it over the allowed range of motion vector displacements. The process is called *block matching*. An exhaustive search guarantees the best possible result, but is computationally expensive, because every possible motion must be tested over the entire displacement range.

Figure 9.21 shows a video encoder that can perform the motion compensated prediction process just described. Think of the input to the encoder as sequential macroblocks of video. The grayed elements parallel the transformation, quantization, and variable-length coding operations of the JPEG encoder in Fig. 9.11(a). The principal difference is the input, which may be a conventional macroblock of image data (e.g., the initial frame to be encoded) or the difference between a conventional macroblock and a prediction of it based on a previous frame (when motion compensation is performed). Note also that the encoder includes an inverse quantizer and inverse DCT so that its predictions match those of the complementary decoder. It also includes a variable-length coder for the computed motion vectors.

Most modern video compression standards (from MPEG-1 to MPEG-4 AVC) can be implemented on an encoder like that in Fig. 9.21. When there is insufficient interframe correlation to make predictive coding effective (even after motion compensation), a block-oriented 2-D transform approach, like JPEG's DCT-based coding, is typically used. Frames that are compressed without a prediction are called *intraframes* or *Independent frames* (*I-frames*). They can be decoded without access to other frames in the video to which they belong. I-frames usually resemble JPEG encoded images and are ideal starting

MPEG is an abreviation for *Motion Pictures Expert Group*, which develops standards that are sanctioned by the *International Sandards Organzation* (ISO) and the *International Electrotechnical Commission* (IEC). AVC is an acronym for advanced video coding.

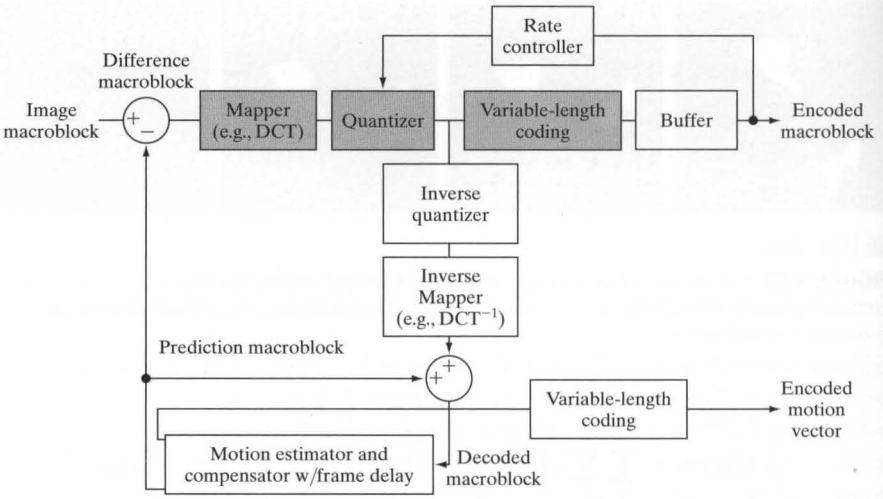


FIGURE 9.21
A typical motion compensated video encoder.

points for the generation of prediction residuals. Moreover, they provide a high degree of random access, ease of editing, and resistance to the propagation of transmission error. As a result, all standards require the periodic insertion of I-frames into the compressed video codestream. An encoded frame that is based on the previous frame is called a *Predictive frame* (*P-frame*); and most standards allow prediction based on a subsequent *Bidirectional frame* (*B-frame*). B-frames require the compressed codestream to be reordered so that frames are presented to the decoder in the proper decoding sequence—rather than the natural display order.

The following function, which we call `tifs2cv`, compresses multiframe TIFF `f` using an exhaustive search strategy with *SAD* as the criterion for selecting the "best" motion vectors. Input `m` determines the size of the macroblocks used (i.e., they are `m x m`), `d` defines the search region (i.e., the maximum macroblock displacement), and `q` sets the quality of the overall compression. If `q` is 0 or omitted, both the prediction residuals and the motion vectors are Huffman coded and the compression is lossless; for all positive nonzero `q`, prediction residuals are coded using `im2jpeg` from Section 9.5.1 and the compression is lossy. Note that the first frame of `f` is treated as an I-frame, while all other frames are coded as P-frames. That is, the code does not perform backward (in time) predictions, nor force the periodic insertion of I-frames that was noted above (and that prevents the buildup of error when using lossy compression). Finally, note that all motion vectors are to the nearest pixel; subpixel interpolations are not performed. The specialized MATLAB block processing functions `im2col` and `col2im`, are used throughout.

`tifs2cv`

```
function y = tifs2cv(f, m, d, q)
%TIFS2CV Compresses a multi-frame TIFF image sequence.
%   Y = TIFS2CV(F, M, D, Q) compresses multiframe TIFF F using
%   motion compensated frames, 8 x 8 DCT transforms, and Huffman
```

```
%   coding. If parameter Q is omitted or is 0, only Huffman
%   encoding is used and the compression is lossless; for Q > 0,
%   lossy JPEG encoding is performed. The inputs are:
%
%      F     A multi-frame TIFF file        (e.g., 'file.tif')
%      M     Macroblock size                (e.g., 8)
%      D     Search displacement            (e.g., [16 8])
%      Q     JPEG quality for IM2JPEG       (e.g., 1)
%
%   Output Y is an encoding structure with fields:
%
%      Y.blksz      Size of motion compensation blocks
%      Y.frames     The number of frames in the image sequence
%      Y.quality    The reconstruction quality
%      Y.motion     Huffman encoded motion vectors
%      Y.video      An array of MAT2HUFF or IM2JPEG coding structures
%
%   See also CV2TIFS.

% The default reconstruction quality is lossless.
if nargin < 4
    q = 0;
end

% Compress frame 1 and reconstruct for the initial reference frame.
if q == 0
    cv(1) = mat2huff(imread(f, 1));
    r = double(huff2mat(cv(1)));
else
    cv(1) = im2jpeg(imread(f, 1), q);
    r = double(jpeg2im(cv(1)));
end
fsz = size(r);

% Verify that image dimensions are multiples of the macroblock size.
if ((mod(fsz(1), m) ~= 0) || (mod(fsz(2), m) ~= 0))
    error('Image dimensions must be multiples of the block size.');
end

% Get the number of frames and preallocate a motion vector array.
fcnt = size(imfinfo(f), 1);
mvsz = [fsz/m 2 fcnt];
mv = zeros(mvsz);

% For all frames except the first, compute motion conpensated
% prediction residuals and compress with motion vectors.
for i = 2:fcnt
    frm = double(imread(f, i));
    frmC = im2col(frm, [m m], 'distinct');
    eC = zeros(size(frmC));

    for col = 1:size(frmC, 2)
        lookfor = col2im(frmC(:,col), [m m], [m m], 'distinct');
```

```
            x = 1 + mod(m * (col - 1), fsz(1));
            y = 1 + m * floor((col - 1) * m / fsz(1));
            x1 = max(1, x - d(1));
            x2 = min(fsz(1), x + m + d(1) - 1);
            y1 = max(1, y - d(2));
            y2 = min(fsz(2), y + m + d(2) - 1);

            here = r(x1:x2, y1:y2);
            hereC = im2col(here, [m m], 'sliding');
            for j = 1:size(hereC, 2)
                hereC(:,j) = hereC(:, j) - lookfor(:);
            end
            sC = sum(abs(hereC));
            s = col2im(sC, [m m], size(here), 'sliding');
            mins = min(min(s));
            [sx sy] = find(s == mins);

            ns = abs(sx) + abs(sy);          % Get the closest vector
            si = find(ns == min(ns));
            n = si(1);

            mv(1 + floor((x - 1)/m), 1 + floor((y - 1)/m), 1:2, i) = ...
                [x - (x1 + sx(n) - 1) y - (y1 + sy(n) - 1)];
            eC(:,col) = hereC(:, sx(n) + (1 + size(here, 1) - m) ...
                * (sy(n) - 1));
        end

        % Code the prediction residual and reconstruct it for use in
        % forming the next reference frame.
        e = col2im(eC, [m m], fsz, 'distinct');
        if q == 0
            cv(i) = mat2huff(int16(e));
            e = double(huff2mat(cv(i)));
        else
            cv(i) = im2jpeg(uint16(e + 255), q, 9);
            e = double(jpeg2im(cv(i)) - 255);
        end

        % Decode the next reference frame. Use the motion vectors to get
        % the subimages needed to subtract from the prediction residual.
        rC = im2col(e, [m m], 'distinct');
        for col = 1:size(rC, 2)
            u = 1 + mod(m * (col - 1), fsz(1));
            v = 1 + m * floor((col - 1) * m / fsz(1));
            rx = u - mv(1 + floor((u - 1)/m), 1 + floor((v - 1)/m), 1, i);
            ry = v - mv(1 + floor((u - 1)/m), 1 + floor((v - 1)/m), 2, i);
            temp = r(rx:rx + m - 1, ry:ry + m - 1);
            rC(:, col) = temp(:) - rC(:, col);
        end
        r = col2im(double(uint16(rC)), [m m], fsz, 'distinct');
    end

    y = struct;
```

```
y.blksz = uint16(m);
y.frames = uint16(fcnt);
y.quality = uint16(q);
y.motion = mat2huff(mv(:));
y.video = cv;
```

Because `tifs2cv` must also decode the encoded prediction residuals that it generates (i.e., they become reference frames in subsequent predictions), it contains most of the code needed to construct a decoder for its output (see the code block beginning with the `rc = im2col(e, [m m], 'distinct')` at the end of the program. Rather than listing the required decoder function here, it is included in Appendix C. The syntax of the function, called `cv2tifs`, is

```
cv2tifs(cv, 'filename.tif')
```

cv2tifs

where `cv` is a `tifs2cv` compressed video sequence and `'filename.tif'` is the multiframe TIFF to which the decompressed output is written. In the following example, we use `tifs2cv`, `cv2tifs`, and custom function `showmo`, which is also listed in Appendx C and whose syntax is

```
v = showmo(cv, indx)
```

showmo

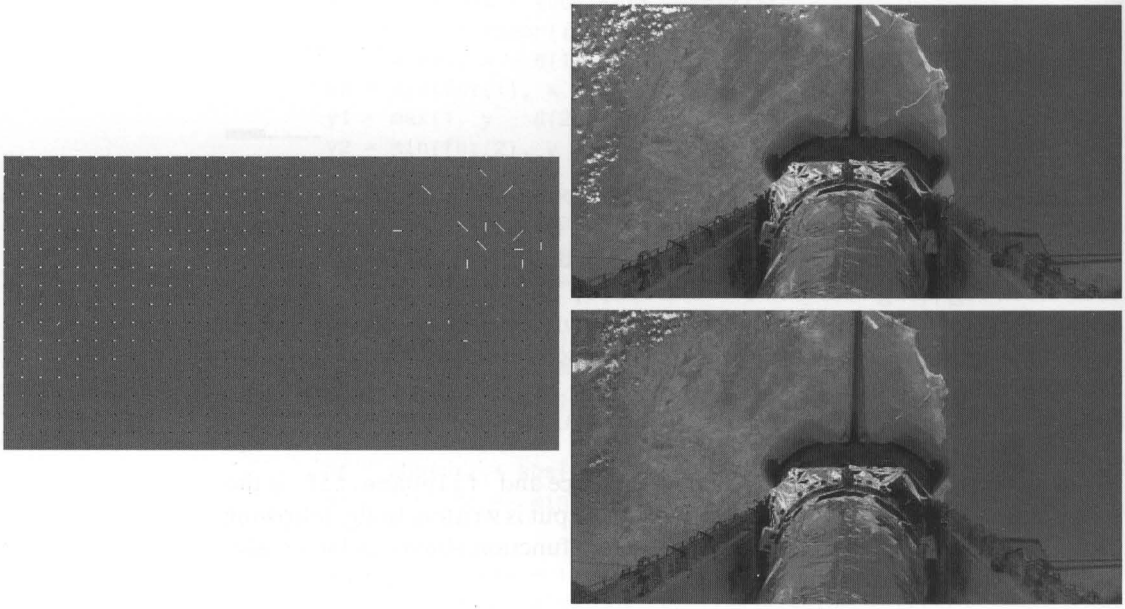
where `v` is a `uint8` image of motion vectors, `cv` is a `tifs2cv` compressed video sequence, and `indx` points to a frame in `cv` whose motion vectors are to be displayed.

EXAMPLE 9.10: Motion compensated video compression.

■ Consider an error-free encoding of the multiframe TIFF whose first and last frames are shown in Fig. 9.18. The following commands perform a lossless motion compensated compression, compute the resulting compression ratio, and display the motion vectors computed for one frame of the compressed sequence:

```
>> cv = tifs2cv('shuttle.tif',16,[8 8]);
>> imratio('shuttle.tif',cv)
ans =
    2.6886
>> showmo(cv, 2);
```

Figure 9.22 shows the motion vectors generated by the `showmo(cv, 2)` statement. These vectors reflect the left-to-right movement of the Earth in the background (see the frames shown in Fig. 9.18) and the lack of motion in the foreground area where the shuttle resides. The black dots in the figure are the heads of the motion vectors and represent the upper-left-hand corners of coded macroblocks. The losslessly compressed video takes only 37% of the memory required to store the original 16-frame uncompressed TIFF.



a b c

FIGURE 9.22 (a) Motion vectors for encoding of the second frame of `'shuttle.tif'`; (b) Frame 2 before encoding and reconstruction; and (c) The reconstructed frame. (Original image coutesy of NASA.)

To increase the compression, we employ a lossy JPEG encoding of the prediction residuals and use the default JPEG normalization array (that is, use `tifs2cv` with input `q` set to 1). The following commands time the compression, decode the compressed video (timing the decompression as well), and compute the rms error of several frames in the reconstructed sequence:

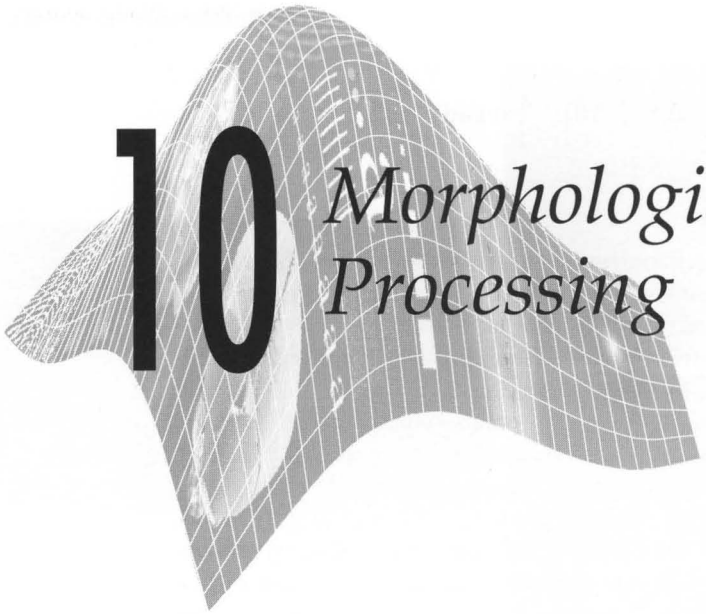
```
>> tic; cv2 = tifs2cv('shuttle.tif', 16, [8 8], 1); toc
Elapsed time is 123.022241 seconds.
>> tic; cv2tifs(cv2, 'ss2.tif'); toc
Elapsed time is 16.100256 seconds.
>> imratio('shuttle.tif', cv2)
ans =
   16.6727
>> compare(imread('shuttle.tif', 1), imread('ss2.tif', 1))
ans =
    6.3368
>> compare(imread('shuttle.tif', 8), imread('ss2.tif', 8))
ans =
```

```
    11.8611
>> compare(imread('shuttle.tif', 16), imread('ss2.tif', 16))
ans =
    14.9153
```

Note that `cv2tifs` (the decompression function) is almost 8 times faster than `tifs2cv` (the compression function)—only 16 seconds as opposed to 123 seconds. This is as should be expected, because the encoder not only performs an exhaustive search for the best motion vectors, (the encoder merely uses those vectors to generate predictions), but decodes the encoded prediction residuals as well. Note also that the rms errors of the reconstructed frames increase from only 6 gray levels for the first frame to almost 15 gray levels for the final frame. Figures 9.22(b) and (c) show an original and reconstructed frame in the middle of the video (i.e., at frame 8). With an rms error of about 12 gray levels, that loss of detail—particularly in the clouds in the upper left and the rivers on the right side of the landmass,—is clearly evident. Finally, we note that with a compression of 16.67 : 1, the motion compensated video uses only 6% of the memory required to store the original uncompressed multiframe TIFF. ■

Summary

The material in this chapter introduces the fundamentals of digital image compression through the removal of coding redundancy, spatial redundancy, temporal redundancy, and irrelevant information. MATLAB routines that attack each of these redundancies—and extend the Image Processing Toolbox—are developed. Both still frame and video coding considered. Finally, an overview of the popular JPEG and JPEG 2000 image compression standards is given. For additional information on the removal of image redundancies—both techniques that are not covered here and standards that address specific image subsets (like binary images)—see Chapter 8 of the third edition of *Digital Image Processing* by Gonzalez and Woods [2008].



10 Morphological Image Processing

Preview

The word *morphology* commonly denotes a branch of biology that deals with the form and structure of animals and plants. We use the same word here in the context of *mathematical morphology* as a tool for extracting image components that are useful in the representation and description of region shape, such as boundaries, skeletons, and the convex hull. We are interested also in morphological techniques for pre- or postprocessing, such as morphological filtering, thinning, and pruning.

In Section 10.1 we define several set theoretic operations and discuss binary sets and logical operators. In Section 10.2 we define two fundamental morphological operations, *dilation* and *erosion*, in terms of the union (or intersection) of an image with a translated shape called a *structuring element*. Section 10.3 deals with combining erosion and dilation to obtain more complex morphological operations. Section 10.4 introduces techniques for labeling connected components in an image. This is a fundamental step in extracting objects from an image for subsequent analysis.

Section 10.5 deals with *morphological reconstruction*, a morphological transformation involving two images, rather than a single image and a structuring element, as is the case in Sections 10.1 through 10.4. Section 10.6 extends morphological concepts to gray-scale images by replacing set union and intersection with maxima and minima. Many binary morphological operations have natural extensions to gray-scale processing. Some, like morphological reconstruction, have applications that are unique to gray-scale images, such as peak filtering.

The material in this chapter begins a transition from image-processing methods whose inputs and outputs are images, to image analysis methods, whose outputs attempt to describe the contents of the image. Morphology is

a cornerstone of the mathematical set of tools underlying the development of techniques that extract "meaning" from an image. Other approaches are developed and applied in the remaining chapters of the book.

10.1 Preliminaries

In this section we introduce some basic concepts from set theory and discuss the application of MATLAB's logical operators to binary images.

10.1.1 Some Basic Concepts from Set Theory

Let Z be the set of real integers. The sampling process used to generate digital images may be viewed as partitioning the xy-plane into a grid, with the coordinates of the center of each grid being a pair of elements from the Cartesian product, Z^2 [†]. In the terminology of set theory, a function $f(x, y)$ is said to be a *digital image* if (x, y) are integers from Z^2 and f is a mapping that assigns an intensity value (that is, a real number from the set of real numbers, R) to each distinct pair of coordinates (x, y). If the elements of R are integers also (as is usually the case in this book), a digital image then becomes a two-dimensional function whose coordinates and amplitude (i.e., intensity) values are integers.

Let A be a set in Z^2, the elements of which are pixel coordinates (x, y). If $w = (x, y)$ is an element of A, then we write

$$w \in A$$

Similarly, if w is not an element of A, we write

$$w \notin A$$

A set B of pixel coordinates that satisfy a particular condition is written as

$$B = \{w | \text{condition}\}$$

For example, the set of all pixel coordinates that do not belong to set A, denoted A^c, is given by

$$A^c = \{w | w \notin A\}$$

This set is called the *complement* of A.

The *union* of two sets, A and B, denoted by

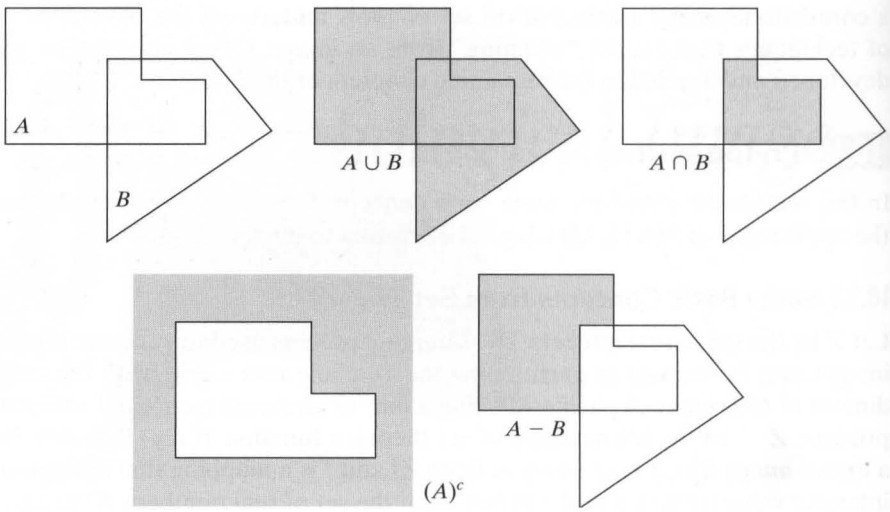
$$C = A \cup B$$

is the set of all elements that belong to A, to B, or to both. Similarly, the *intersection* of sets A and B, denoted by

$$C = A \cap B$$

is the set of all elements that belong to both A and B.

[†] The Cartesian product of a set of integers, Z, is the set of all ordered pairs of elements (z_i, z_j), with z_i and z_j being integers from Z. It is customary to denote the Cartesian product by Z^2.



a b c
d e

FIGURE 10.1
(a) Two sets A and B. (b) The union of A and B. (c) The intersection of A and B. (d) The complement of A. (e) The difference between A and B.

The *difference* of sets A and B, denoted $A - B$, is the set of all elements that belong to A but not to B:

$$A - B = \{w \mid w \in A, w \notin B\}$$

Figure 10.1 illustrates the set operations defined thus far. The result of each operation is shown in gray.

In addition to the preceding basic operations, morphological operations often require two operators that are specific to sets whose elements are pixel coordinates. The *reflection* of a set B, denoted $\hat{B}$, is defined as

$$\hat{B} = \{w \mid w = -b \text{ for } b \in B\}$$

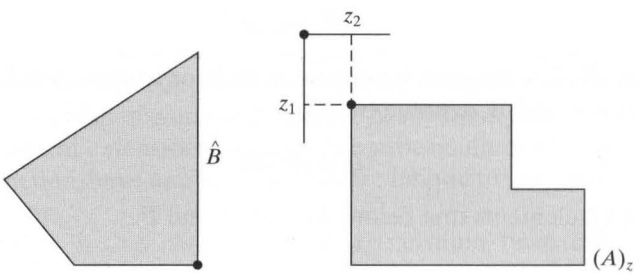
The *translation* of set A by point $z = (z_1, z_2)$, denoted $(A)_z$, is defined as

$$(A)_z = \{c \mid c = a + z \text{ for } a \in A\}$$

Figure 10.2 illustrates these two definitions using the sets from Fig. 10.1. The black dot denotes the *origin* of the sets (the origin is a user-defined reference point).

a b

FIGURE 10.2
(a) Reflection of B. (b) Translation of A by z. The sets A and B are from Fig. 10.1, and the black dot denotes their origin.



10.1.2 Binary Images, Sets, and Logical Operators

The language and theory of mathematical morphology often present a dual (but equivalent) view of binary images. Thus far, we have considered a binary image to be a bivalued *function* of spatial coordinates x and y. Morphological theory views a binary image as a *set* of *foreground* (1-valued) pixels, the elements of which are in Z^2. Set operations such as union and intersection can be applied directly to binary image sets. For example, if A and B are binary images, then $C = A \cup B$ is a binary image also, where a pixel in C is a foreground pixel if either or both of the corresponding pixels in A and B are foreground pixels. In the first view, that of a function, C is given by

$$C(x, y) = \begin{cases} 1 & \text{if either } A(x, y) \text{ or } B(x, y) \text{ is 1, or if both are 1} \\ 0 & \text{otherwise} \end{cases}$$

On the other hand, using the set point of view, C is given by

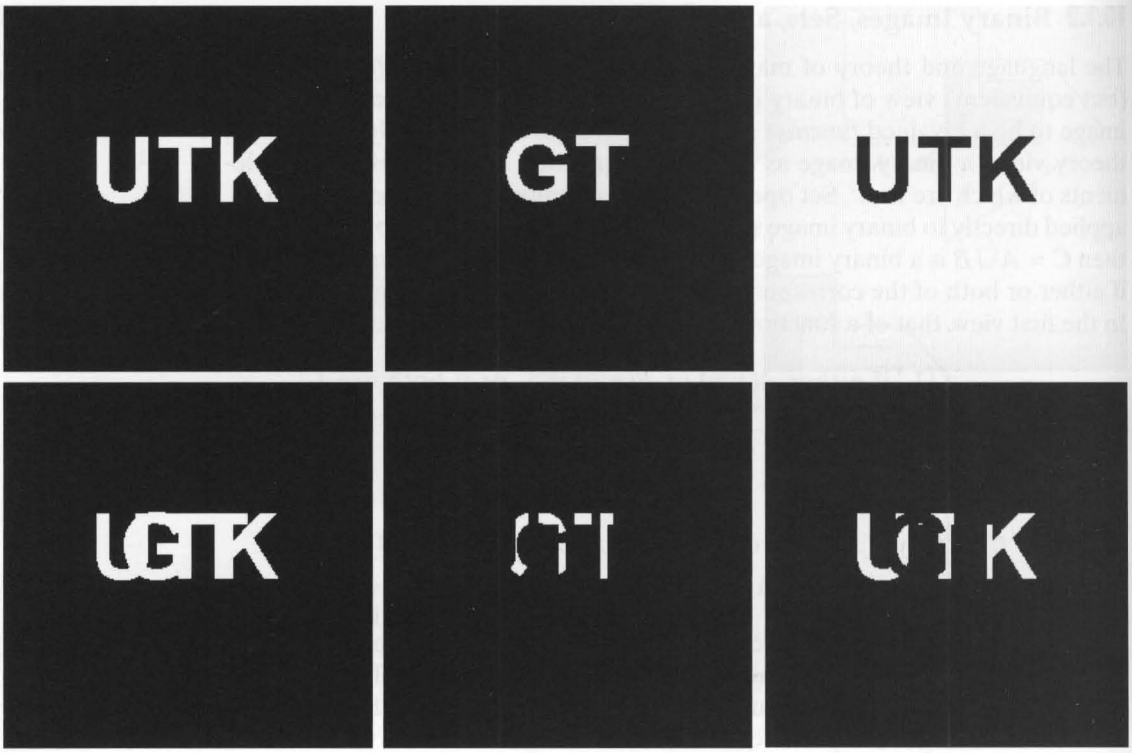
$$C = \{(x, y) \mid (x, y) \in A \text{ or } (x, y) \in B \text{ or } (x, y) \in (A \text{ and } B)\}$$

where, as mentioned previously regarding the set point of view, the elements of A and B are 1-valued. Thus, we see that the function point of view deals with both foreground (1) and background (0) pixels simultaneously. The set point of view deals only with foreground pixels, and it is understood that all pixels that are not foreground pixels constitute the *background*. Of course, results using either point of view are the same. The set operations defined in Fig. 10.1 can be performed on *binary* images using MATLAB's logical operators OR (|), AND (&), and NOT (~), as Table 10.1 shows.

As an illustration, Fig. 10.3 shows the results of applying several logical operators to two binary images containing text. (We follow the Image Processing Toolbox convention that foreground (1-valued) pixels are displayed as white.) The image in Fig. 10.3(d) is the union of the "UTK" and "GT" images; it contains all the foreground pixels from both. In contrast, the intersection of the two images [Fig. 10.3(e)] shows the pixels where the letters in "UTK" and "GT" overlap. Finally, the set difference image [Fig. 10.3(f)] shows the letters in "UTK" with the pixels "GT" removed.

Set Operation	MATLAB Expression for Binary Images	Name
$A \cap B$	A & B	AND
$A \cup B$	A | B	OR
A^c	~A	NOT
$A - B$	A & ~B	DIFFERENCE

TABLE 10.1 Using logical expressions in MATLAB to perform set operations on binary images.



a b c
d e f

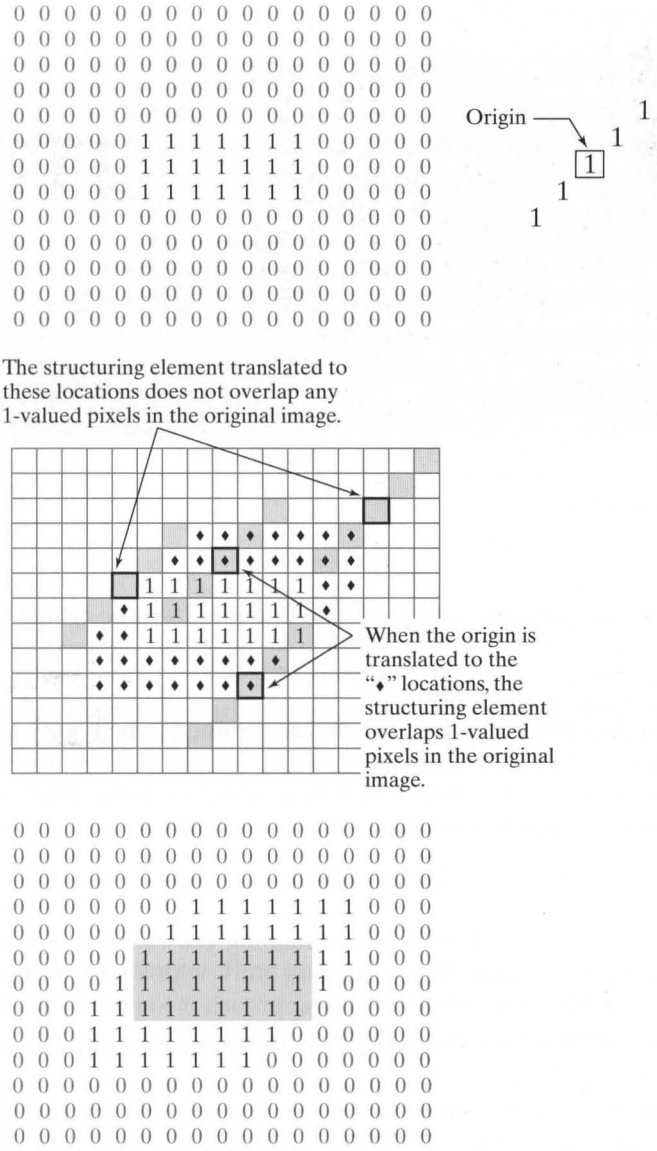
FIGURE 10.3 (a) Binary image A. (b) Binary image B. (c) Complement ~A. (d) Union A | -B. (e) Intersection A & B. (f) Set difference A & ~B.

10.2 Dilation and Erosion

The operations of *dilation* and *erosion* are fundamental to morphological image processing. Many of the algorithms presented later in this chapter are based on these operations.

10.2.1 Dilation

Dilation is an operation that "grows" or "thickens" objects in an image. The specific manner and extent of this thickening is controlled by a shape referred to as a *structuring element*. Figure 10.4 illustrates how dilation works. Figure 10.4(a) shows a binary image containing a rectangular object. Figure 10.4(b) is a structuring element, a five-pixel-long diagonal line in this case. Graphically, structuring elements can be are represented either by a matrix of 0s and 1s or as a set of foreground (1-valued) pixels, as in Fig. 10.4(b). We use both representations interchangeably in this chapter. Regardless of the representation, the origin of the structuring element must be clearly identified. Figure 10.4(b) indicates the



a b
c
d

FIGURE 10.4 Illustration of dilation. (a) Original image with rectangular object. (b) Structuring element with five pixels arranged in a diagonal line. The origin, or center, of the structuring element is shown with a dark border. (c) Structuring element translated to several locations in the image. (d) Output image. The shaded region shows the location of 1s in the original image.

origin of the structuring element by a black box. Figure 10.4(c) depicts dilation as a process that translates the origin of the structuring element throughout the domain of the image and checks to see where the element overlaps 1-valued pixels. The output image [Fig. 10.4(d)] is 1 at each *location* of the *origin* of the structuring element such that the structuring element overlaps at least one 1-valued pixel in the input image.

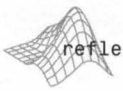
The dilation of A by B, denoted $A \oplus B$, is defined as the set operation

$$A \oplus B = \{z \mid (\hat{B})_z \cap A \neq \varnothing\}$$

You can see here an example of the importance of the origin of a structuring element. Changing the location of the defined origin generally changes the result of a morphological operation.

where $\varnothing$ is the empty set and B is the structuring element. In words, the dilation of A by B is the set consisting of all the structuring element *origin locations* where the reflected and translated B overlaps at least one element of A. It is a convention in image processing to let the first operand of $A \oplus B$ be the image and the second operand be the structuring element, which usually is much smaller than the image. We follow this convention from this point on.

The translation of the structuring element in dilation is similar to the mechanics of spatial convolution discussed in Chapter 3. Figure 10.4 does not show the structuring element's reflection explicitly because the structuring element is symmetrical with respect to its origin in this case. Figure 10.5 shows a nonsymmetric structuring element and its reflection. Toolbox function `reflect` can be used to compute the reflection of a structuring element.



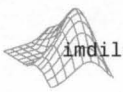
Dilation is *associative*,

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C$$

and *commutative*:

$$A \oplus B = B \oplus A$$

Toolbox function `imdilate` performs dilation. Its basic calling syntax is




```
D = imdilate(A, B)
```

For the moment, the inputs and output are assumed to be binary, but the same syntax can deal with gray-scale functions, as discussed in Section 10.6. Assuming binary quantities for now, `B` is a structuring element array of 0s and 1s whose origin is computed *automatically* by the toolbox as

```
floor((size(B) + 1)/2)
```

This operation yields a 2-D vector containing the coordinates of the center of the structuring element. If you need to work with a structuring element in which the origin is *not in the center*, the approach is to pad `B` with zeros so that the original center is shifted to the desired location.

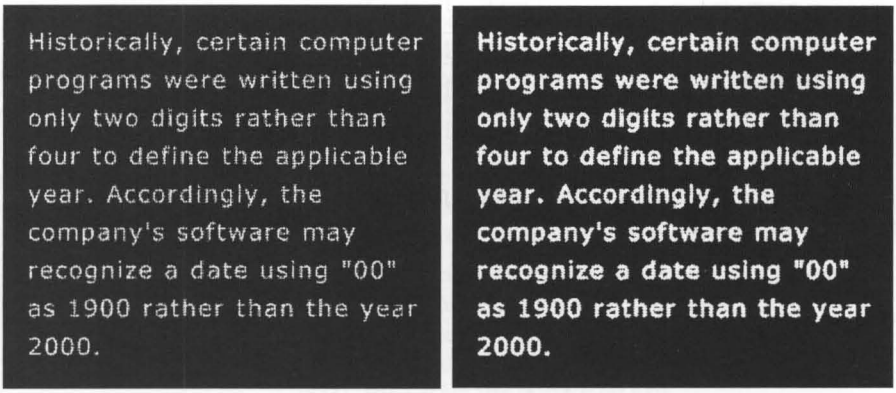
EXAMPLE 10.1: An application of dilation.

■ Figure 10.6(a) shows a binary image containing text with numerous broken characters. We want to use `imdilate` to dilate the image with the following structuring element:



a b

FIGURE 10.5 (a) Nonsymmetric structuring element. (b) Structuring element reflected about its origin.



a b

FIGURE 10.6
An example of dilation.
(a) Input image containing broken text.
(b) Dilated image.

$$\begin{matrix} 0 & 1 & 0 \\ 1 & \boxed{1} & 1 \\ 0 & 1 & 0 \end{matrix}$$

The following commands read the image from a file, form the structuring element matrix, perform the dilation, and display the result.

```
>> A = imread('broken_text.tif');
>> B = [0 1 0; 1 1 1; 0 1 0];
>> D = imdilate(A, B);
>> imshow(D)
```

Figure 10.6(b) shows the resulting image. ■

10.2.2 Structuring Element Decomposition

Suppose that a structuring element B can be represented as a dilation of two structuring elements B_1 and B_2:

$$B = B_1 \oplus B_2$$

Then, because dilation is associative, $A \oplus B = A \oplus (B_1 \oplus B_2) = (A \oplus B_1) \oplus B_2$. In other words, dilating A with B is the same as first dilating A with B_1 and then dilating the result with B_2. We say that B can be *decomposed* into the structuring elements B_1 and B_2.

The associative property is important because the time required to compute dilation is proportional to the number of nonzero pixels in the structuring element. Consider, for example, dilation with a 5×5 array of 1s:

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \boxed{1} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$$

This structuring element can be decomposed into a five-element row of 1s and a five-element column of 1s:

$$\begin{bmatrix} 1 & 1 & \boxed{1} & 1 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ \boxed{1} \\ 1 \\ 1 \end{bmatrix}$$

The number of elements in the original structuring element is 25, but the total number of elements in the row-column decomposition is only 10. This means that dilation with the row structuring element first, followed by dilation with the column element, can be performed 2.5 times faster than dilation with the 5×5 array of 1s. In practice, the speed-up will be somewhat less because usually there is some overhead associated with each dilation operation. However, the gain in speed with the decomposed implementation is still significant.

10.2.3 The `strel` Function

Toolbox function `strel` constructs structuring elements with a variety of shapes and sizes. Its basic syntax is




```
se = strel(shape, parameters)
```

where `shape` is a string specifying the desired shape, and `parameters` is a list of parameters that specify information about the shape, such as its size. For example, `strel('diamond', 5)` returns a diamond-shaped structuring element that extends ± 5 pixels along the horizontal and vertical axes. Table 10.2 summarizes the various shapes that `strel` can create.

In addition to simplifying the generation of common structuring element shapes, function `strel` also has the important property of producing structuring elements in decomposed form. Function `imdilate` automatically uses the decomposition information to speed up the dilation process. The following example illustrates how `strel` returns information related to the decomposition of a structuring element.

Syntax Form	Description
`se = strel('diamond', R)`	Creates a flat, diamond-shaped structuring element, where `R` specifies the distance from the structuring element origin to the extreme points of the diamond.
`se = strel('disk', R)`	Creates a flat, disk-shaped structuring element with radius `R`. (Additional parameters may be specified for the disk; see the `strel` reference page for details.)
`se = strel('line', LEN, DEG)`	Creates a flat, linear structuring element, where `LEN` specifies the length, and `DEG` specifies the angle (in degrees) of the line, as measured in a counterclockwise direction from the horizontal axis.
`se = strel('octagon', R)`	Creates a flat, octagonal structuring element, where `R` specifies the distance from the structuring element origin to the sides of the octagon, as measured along the horizontal and vertical axes. `R` must be a nonnegative multiple of 3.
`se = strel('pair', OFFSET)`	Creates a flat structuring element containing two members. One member is located at the origin. The location of the second member is specified by the vector `OFFSET`, which must be a two-element vector of integers.
`se = strel('periodicline', P, V)`	Creates a flat structuring element containing `2*P+1` members; `V` is a two-element vector containing integer-valued row and column offsets. One structuring element member is located at the origin. The other members are located at `1*V`, `-1*V`, `2*V`, `-2*V`, `...`, `P*V`, and `-P*V`.
`se = strel('rectangle', MN)`	Creates a flat, rectangle-shaped structuring element, where `MN` specifies the size. `MN` must be a two-element vector of nonnegative integers. The first element of `MN` is the number of rows in the structuring element; the second element is the number of columns.
`se = strel('square', W)`	Creates a square structuring element whose width is `W` pixels. `W` must be a nonnegative integer.
`se = strel('arbitrary', NHOOD)` `se = strel(NHOOD)`	Creates a structuring element of arbitrary shape. `NHOOD` is a matrix of 0s and 1s that specifies the shape. The second, simpler syntax form shown performs the same operation.

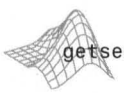
TABLE 10.2 The various syntax forms of function `strel`. The word *flat* indicates two-dimensional structuring elements (i.e., elements of zero height). This qualifier is meaningful in the context of gray-scale dilation and erosion, as discussed in Section 10.6.1.

EXAMPLE 10.2: Structuring element decomposition using function `strel`.

■ Consider the creation of a diamond-shaped structuring element using function `strel`:

```
>> se = strel('diamond', 5)
se =
Flat STREL object containing 61 neighbors.
Decomposition: 4 STREL objects containing a total of 17
neighbors
Neighborhood:

     0     0     0     0     0     1     0     0     0     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     1     1     1     1     1     0     0     0
     0     0     1     1     1     1     1     1     1     0     0
     0     1     1     1     1     1     1     1     1     1     0
     1     1     1     1     1     1     1     1     1     1     1
     0     1     1     1     1     1     1     1     1     1     0
     0     0     1     1     1     1     1     1     1     0     0
     0     0     0     1     1     1     1     1     0     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     0     0     1     0     0     0     0     0
```

The output of function `strel` is not a normal MATLAB matrix; instead, it is a special kind of quantity called an *strel object*. The command-window display of an strel object includes the neighborhood (a matrix of 1s in a diamond-shaped pattern in this case); the number of 1-valued pixels in the structuring element (61); the number of structuring elements in the decomposition (4); and the total number of 1-valued pixels in the decomposed structuring elements (17). Function `getsequence` can be used to extract and examine separately the individual structuring elements in the decomposition.




```
>> decomp = getsequence(se);
>> whos
  Name          Size                Bytes  Class    Attributes
  decomp        4x1                  1716  strel
  se            1x1                  3309  strel
```

The output of `whos` shows that `se` and `decomp` are both strel objects and, further, that `decomp` is a four-element vector of strel objects. The four structuring elements in the decomposition can be examined individually by indexing into `decomp`:

```
>> decomp(1)
ans =
Flat STREL object containing 5 neighbors.
```

```
Neighborhood:
     0     1     0
     1     1     1
     0     1     0

>> decomp(2)
ans =
Flat STREL object containing 4 neighbors.
Neighborhood:
     0     1     0
     1     0     1
     0     1     0


>> decomp(3)
ans =
Flat STREL object containing 4 neighbors.
Neighborhood:
     0     0     1     0     0
     0     0     0     0     0
     1     0     0     0     1
     0     0     0     0     0
     0     0     1     0     0


>> decomp(4)
ans =
Flat STREL object containing 4 neighbors.
Neighborhood:
     0     1     0
     1     0     1
     0     1     0
```

Function `imdilate` uses the decomposed form of a structuring element automatically, performing dilation approximately three times faster ($\approx 61/17$) in this case than with the non-decomposed form. ■

10.2.4 Erosion

Erosion "shrinks" or "thins" objects in a binary image. As in dilation, the manner and extent of shrinking is controlled by a structuring element. Figure 10.7 illustrates the erosion process. Figure 10.7(a) is the same as Fig. 10.4(a). Figure

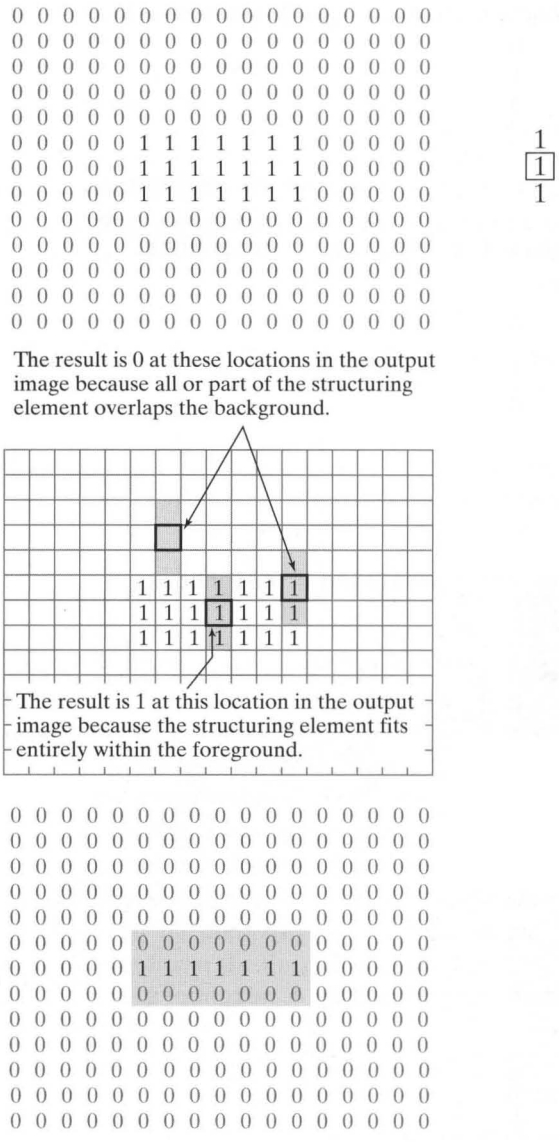


FIGURE 10.7 Illustration of erosion. (a) Original image with rectangular object. (b) Structuring element with three pixels arranged in a vertical line. The origin of the structuring element is shown with a dark border. (c) Structuring element translated to several locations in the image. (d) Output image. The shaded region shows the location of 1s in the original image.

10.7(b) is the structuring element, a short vertical line. Figure 10.7(c) depicts erosion graphically as a process of translating the structuring element throughout the domain of the image and checking to see where it fits entirely within the foreground of the image. The output image in Fig. 10.7(d) has a value of 1 at each location of the *origin* of the structuring element, such that the element overlaps *only* 1-valued pixels of the input image (i.e., it does not overlap any of the image background).

The erosion of A by B, denoted $A \ominus B$, is defined as

$$A \ominus B = \{z | (B)_z \subseteq A\}$$

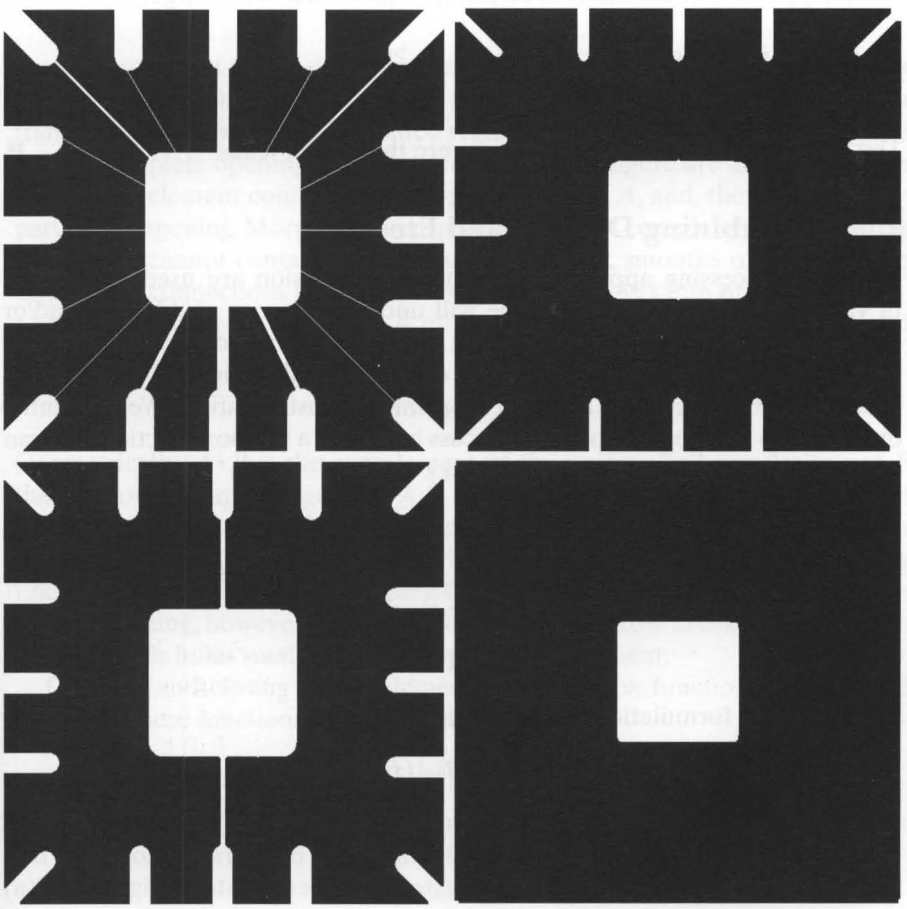
where, as usual, the notation $C \subseteq D$ means that C is a subset of D. This equation says that the erosion of A by B is the set of all points z such that B, translated by z, is contained in A. Because the statement that B is contained in A is equivalent to B not sharing any elements with the background of A, we can write the following equivalent expression as the definition of erosion:

$$A \ominus B = \{z | (B)_z \cap A^c = \varnothing\}$$

Here, erosion of A by B is the set of all structuring element *origin locations* where no part of B overlaps the background of A.

EXAMPLE 10.3: An illustration of erosion.

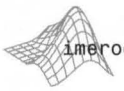
■ Erosion is performed by toolbox function `imerode`, whose syntax is the same as the syntax of `imdilate` discussed in Section 10.2.1. Suppose that we want to remove the thin wires in the binary image in Fig. 10.8(a), while



a b
c d

FIGURE 10.8 An illustration of erosion. (a) Original image of size 486 × 486 pixels. (b) Erosion with a disk of radius 10. (c) Erosion with a disk of radius 5. (d) Erosion with a disk of radius 20.

preserving the other structures. We can do this by choosing a structuring element small enough to fit within the center square and thicker border leads but too large to fit entirely within the wires. Consider the following commands:




```
>> A = imread('wirebond_mask.tif');
>> se = strel('disk', 10);
>> E10 = imerode(A, se);
>> imshow(E10)
```

As Fig. 10.8(b) shows, these commands successfully removed the thin wires in the mask. Figure 10.8(c) shows what happens if we choose a structuring element that is too small:

```
>> se = strel('disk', 5);
>> E5 = imerode(A, se);
>> imshow(E5)
```

Some of the wire leads were not removed in this case. Figure 10.8(d) shows what happens if we choose a structuring element that is too large:

```
>> E20 = imerode(A, strel('disk', 20));
>> imshow(E20)
```

The wire leads were removed, but so were the border leads. ■

10.3 Combining Dilation and Erosion

In image-processing applications, dilation and erosion are used most often in various combinations. An image will undergo a series of dilations and/or erosions using the same, or sometimes different, structuring elements. In this section we consider three of the most common combinations of dilation and erosion: opening, closing, and the hit-or-miss transformation. We also introduce lookup table operations and discuss `bwmorph`, a toolbox function that can perform a variety of morphological tasks.

10.3.1 Opening and Closing

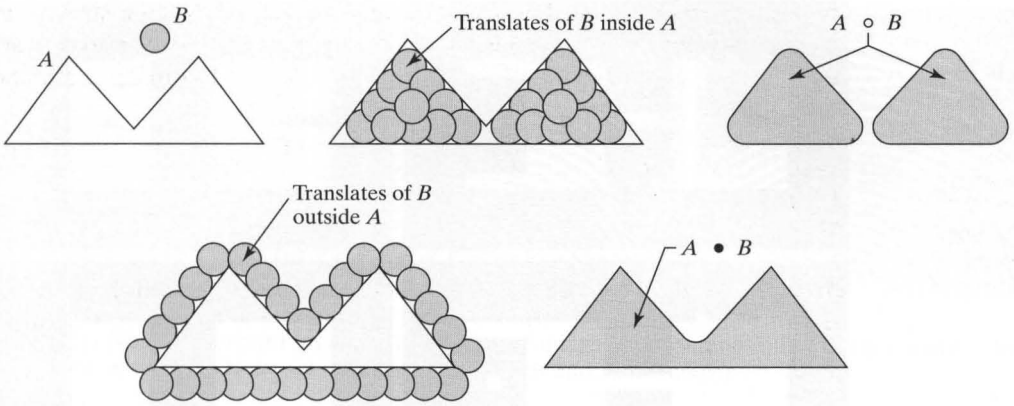
The *morphological opening* of A by B, denoted $A \circ B$, is defined as the erosion of A by B, followed by a dilation of the result by B:

$$A \circ B = (A \ominus B) \oplus B$$

An equivalent formulation of opening is

$$A \circ B = \cup\{(B)_z \mid (B)_z \subseteq A\}$$

where $\cup\{\cdot\}$ denotes the union of all sets inside the braces. This formulation has a simple geometric interpretation: $A \circ B$ is the union of all translations of B that fit entirely within A. Figure 10.9 illustrates this interpretation. Figure 10.9(a)



a b c
d e

FIGURE 10.9 Opening and closing as unions of translated structuring elements. (a) Set A and structuring element B. (b) Translations of B that fit entirely within set A. (c) The complete opening (shaded). (d) Translations of B outside the border of A. (e) The complete closing (shaded).

shows a set A and a disk-shaped structuring element, B. Figure 10.9(b) shows some of the translations of B that fit *entirely* within A. The union of all such translations results in the two shaded regions in Fig. 10.9(c); these two regions are the complete opening. The white regions in this figure are areas where the structuring element could not fit completely within A, and, therefore, are not part of the opening. Morphological opening removes completely regions of an object that cannot contain the structuring element, smooths object contours, breaks thin connections [as in Fig. 10.9(c)], and removes thin protrusions.

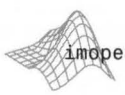
The *morphological closing* of A by B, denoted $A \bullet B$, is a dilation followed by an erosion:

$$A \bullet B = (A \oplus B) \ominus B$$

Geometrically, $A \bullet B$ is the complement of the union of all translations of B that do not overlap A. Figure 10.9(d) illustrates several translations of B that do not overlap A. By taking the complement of the union of all such translations, we obtain the shaded region if Fig. 10.9(e), which is the complete closing. Like opening, morphological closing tends to smooth the contours of objects. Unlike opening, however, closing generally joins narrow breaks, fills long thin gulfs, and fills holes smaller than the structuring element.

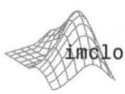
Opening and closing are implemented by toolbox functions `imopen` and `imclose`. These functions have the syntax forms

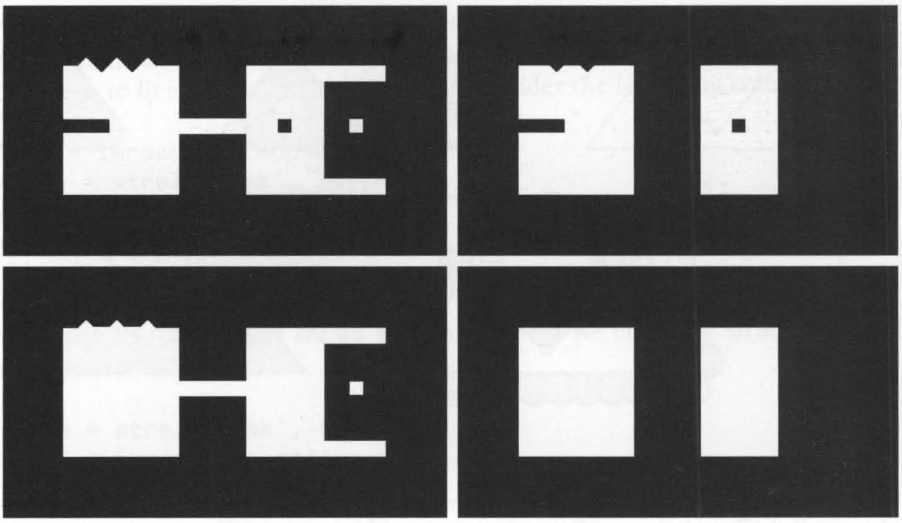
```
C = imopen(A, B)
```



and

```
C = imclose(A, B)
```





a b
c d

FIGURE 10.10
Illustration of opening and closing.
(a) Original image.
(b) Opening.
(c) Closing.
(d) Closing of (b).

where, for now, `A` is a binary image and `B` is a matrix of 0s and 1s that specifies the structuring element. An strel object from Table 10.2 can be used instead of `B`.

EXAMPLE 10.4: Working with functions `imopen` and `imclose`.

■ This example illustrates the use of functions `imopen` and `imclose`. The image `shapes.tif` shown in Fig. 10.10(a) has several features designed to illustrate the characteristic effects of opening and closing, such as thin protrusions, a thin bridge, several gulfs, an isolated hole, a small isolated object, and a jagged boundary. The following commands open the image with a 20×20 square structuring element:

```
>> f = imread('shapes.tif');
>> se = strel('square', 20);
>> fo = imopen(f, se);
>> imshow(fo)
```

Figure 10.10(b) shows the result. Note that the thin protrusions and outward-pointing boundary irregularities were removed. The thin bridge and the small isolated object were removed also. The commands

```
>> fc = imclose(f, se);
>> imshow(fc)
```

produced the result in Fig. 10.10(c). Here, the thin gulf, the inward-pointing boundary irregularities, and the small hole were removed. Closing the result of the earlier opening has a smoothing effect:

```
>> foc = imclose(fo, se);
```



a b c

FIGURE 10.11 (a) Noisy fingerprint image. (b) Opening of image. (c) Opening followed by closing. (Original image courtesy of the U. S. National Institute of Standards and Technology.)

```
>> imshow(foc)
```

Figure 10.10(d) shows the resulting smoothed objects.

An opening/closing sequence can be used for noise reduction. As an example, consider Figure 10.11(a), which shows a noisy fingerprint. The commands

```
>> f = imread('Fig1011(a)

.tif');
>> se = strel('square', 3);
>> fo = imopen(f, se);
>> imshow(fo)
```

produced the image in Fig. 10.11(b). The noisy spots were removed by opening the image, but this process introduced numerous gaps in the ridges of the fingerprint. Many of the gaps can be bridged by following the opening with a closing:

```
>> foc = imclose(fo,se);
>> imshow(foc)
```

Figure 10.11(c) shows the final result, in which most of the noise was removed (at the expense of introducing some gaps in the fingerprint ridges). ■

10.3.2 The Hit-or-Miss Transformation

Often, it is useful to be able to match specified configurations of pixels in an image, such as isolated foreground pixels, or pixels that are endpoints of line segments. The *hit-or-miss transformation* is useful for applications such as these. The hit-or-miss transformation of A by B is denoted $A \otimes B$, where B is a structuring element pair, $B = (B_1, B_2)$, rather than a single element, as before. The hit-or-miss transformation is defined in terms of these two structuring elements as

See *matching* in the Index for other approaches to object matching.

a b
c
d e
f
g

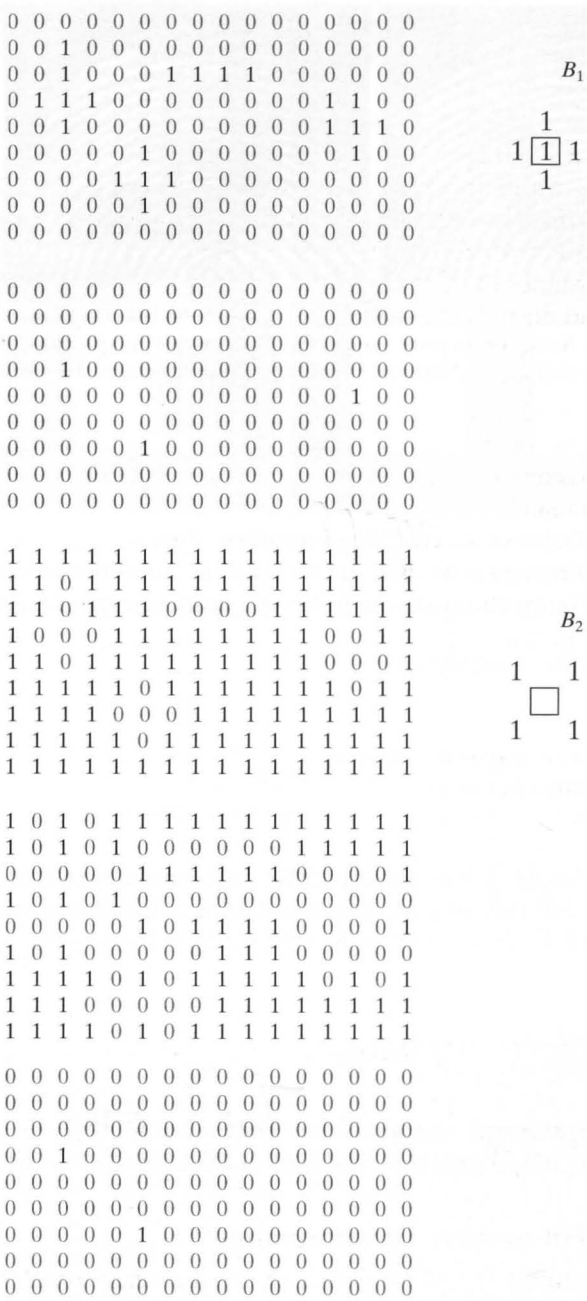


FIGURE 10.12
(a) Original image A. (b) Structuring element B_1. (c) Erosion of A by B_1. (d) Complement of the original image, A^c. (e) Structuring element B_2. (f) Erosion of A^c by B_2. (g) Output image.

$$A \otimes B = (A \ominus B_1) \cap (A^c \ominus B_2)$$

Figure 10.12 illustrates how the hit-or-miss transformation can be used to identify the locations of the following cross-shaped pixel configuration:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}$$

Figure 10.12(a) contains this configuration of pixels in two different locations. Erosion with structuring element B_1 determines the locations of foreground pixels that have north, east, south, and west foreground neighbors [Fig. 10.12(c)]. Erosion of the complement of Fig. 10.12(a) with structuring element B_2 determines the locations of all the pixels whose northeast, southeast, southwest, and northwest neighbors belong to the background [Fig. 10.12(f)]. Figure 10.12(g) shows the intersection (logical AND) of these two operations. Each foreground pixel of Fig. 10.12(g) is the location of the center of a set of pixels having the desired configuration.

The name "hit-or-miss transformation" is based on how the result is affected by the two erosions. For example, the output image in Fig. 10.12 consists of all locations that match the pixels in B_1 (a "hit") and that have none of the pixels in B_2 (a "miss"). Strictly speaking, the term *hit-and-miss transformation* is more accurate, but *hit-or-miss transformation* is used more frequently.

The hit-or-miss transformation is implemented by toolbox function `bwhitmiss`, which has the syntax

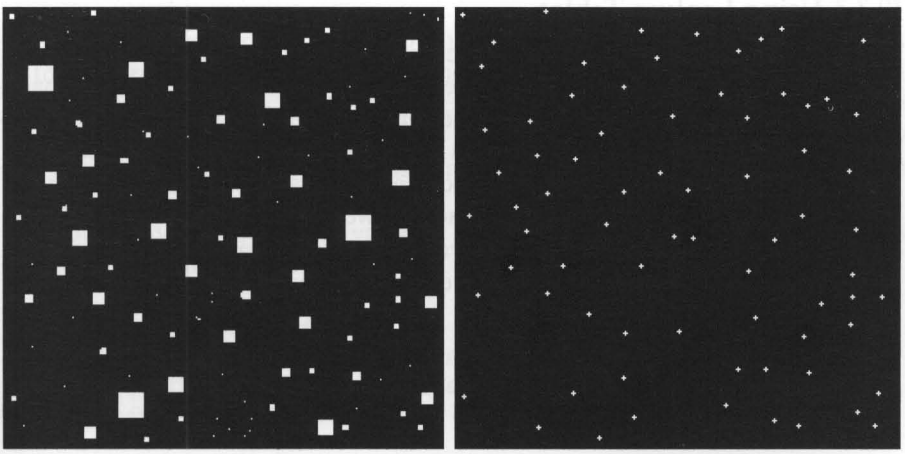
```
C = bwhitmiss(A, B1, B2)
```



where `C` is the result, `A` is the input image, and `B1` and `B2` are the structuring elements just discussed.

EXAMPLE 10.5: Using function `bwhitmiss`.

■ Consider the task of locating upper-left-corner pixels of square objects in an image using the hit-or-miss transformation. Figure 10.13(a) shows an image containing squares of various sizes. We want to locate foreground pixels that



a b

FIGURE 10.13 (a) Original image. (b) Result of applying the hit-or-miss transformation (the dots shown were enlarged to facilitate viewing).

have east and south neighbors (these are "hits") and that have no northeast, north, northwest, west, or southwest neighbors (these are "misses"). These requirements lead to the two structuring elements:

```
>> B1 = strel([0 0 0; 0 1 1; 0 1 0]);
>> B2 = strel([1 1 1; 1 0 0; 1 0 0]);
```

Note that neither structuring element contains the southeast neighbor, which is called a *don't care* pixel. We use function `bwhitmiss` to compute the transformation, where `A` is the input image shown in Fig. 10.13(a):

```
>> C = bwhitmiss(A, B1, B2);
>> imshow(C)
```

Each single-pixel dot in Fig. 10.13(b) is an upper-left-corner pixel of the objects in Fig. 10.13(a). The pixels in Fig. 10.13(b) were enlarged for clarity.

An alternate syntax for `bwhitmiss` combines `B1` and `B2` into an *interval matrix*. The interval matrix equals 1 wherever `B1` equals 1, and is −1 wherever `B2` equals 1. For *don't care* pixels, the interval matrix equals 0. The interval matrix corresponding to `B1` and `B2` above is:

```
>> interval = [-1 -1 -1; -1 1 1; -1 1 0]
interval =
    -1    -1    -1
    -1     1     1
    -1     1     0
```

With this interval matrix, the output image, `C`, can be computed using the syntax `bwhitmiss(A, interval)`. ■

10.3.3 Using Lookup Tables

When the hit-or-miss structuring elements are small, a faster way to compute the hit-or-miss transformation is to use a lookup table (LUT). The approach is to precompute the output pixel value for every possible neighborhood configuration and then store the answers in a table for later use. For instance, there are $2^9 = 512$ different 3×3 configurations of pixel values in a binary image.

To make the use of lookup tables practical, we must assign a *unique* index to each possible configuration. A simple way to do this for, say, the 3×3 case, is to multiply each 3×3 configuration elementwise by the matrix

See Section 2.10.2 for a definition of elementwise operations.

$$\begin{matrix} 1 & 8 & 64 \\ 2 & 16 & 128 \\ 4 & 32 & 256 \end{matrix}$$

and then sum all the products. This procedure assigns a unique value in the range [0, 511] to each different 3×3 neighborhood configuration. For example, the value assigned to the neighborhood

$$\begin{matrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{matrix}$$

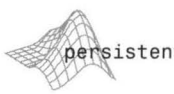
is 1(1) + 2(1) + 4(1) + 8(1) + 16(0) + 32(0) + 64(0) + 128(1) + 256(1) = 399, where the first number in these products is a coefficient from the preceding matrix and the numbers in parentheses are the pixel values, taken column-wise.



The Image Processing Toolbox provides two functions, `makelut` and `applylut` (illustrated later in this section), that can be used to implement this technique. Function `makelut` constructs a lookup table based on a user-supplied function, and `applylut` processes binary images using this lookup table. Continuing with the 3×3 case, using `makelut` requires writing a function that accepts a 3×3 binary matrix and returns a single value, typically either a 0 or 1. Function `makelut` calls the user-supplied function 512 times, passing it each possible 3×3 neighborhood configuration, and returns all the results in the form of a 512-element vector.

As an illustration, we write a function, `endpoints.m`, that uses `makelut` and `applylut` to detect end points in a binary image. We define an *end point* as a foreground pixel whose neighbor configuration matches the hit-or-miss interval matrix `[0 1 0; -1 1 -1; -1 -1 -1]` or any of its 90-degree rotations; or a foreground pixel whose neighbor configuration matches the hit-or-miss interval matrix `[1 -1 -1; -1 1 -1; -1 -1 -1]` or any of its 90-degree rotations (Gonzalez and Woods [2008]). Function `endpoints` computes and then applies a lookup table for detecting end points in an input image. The line of code

```
persistent lut
```



used in function `endpoints` establishes a variable called `lut` and declares it to be *persistent*. MATLAB remembers the value of persistent variables in between function calls. The first time function `endpoints` is called, variable `lut` is initialized automatically to the empty matrix, [ ]. When `lut` is empty, the function calls `makelut`, passing it a handle to subfunction `endpoint_fcn`. Function `applylut` then finds the end points using the lookup table. The lookup table is saved in persistent variable `lut` so that, the next time `endpoints` is called, the lookup table does not need to be recomputed.

```
function g = endpoints(f)
%ENDPOINTS Computes end points of a binary image.
%   G = ENDPOINTS(F) computes the end points of the binary image F
%   and returns them in the binary image G.
```

endpoints

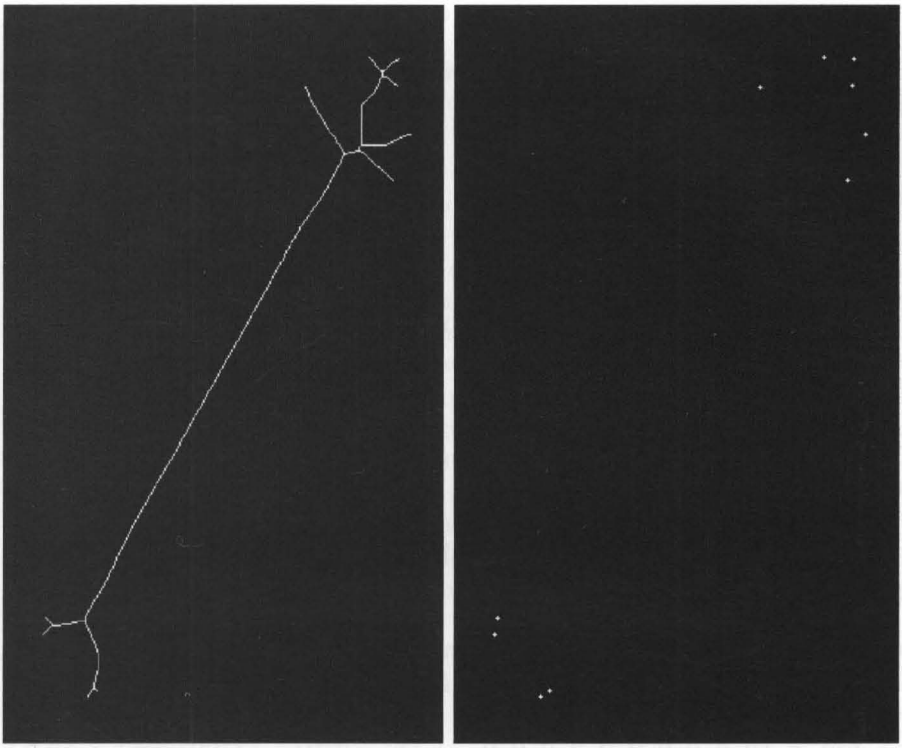
```
persistent lut

if isempty(lut)
   lut = makelut(@endpoint_fcn, 3);
end

g = applylut(f,lut);

%-----------------------------------------------------------------%
function is_end_point = endpoint_fcn(nhood)
%   Determines if a pixel is an end point.
%   IS_END_POINT = ENDPOINT_FCN(NHOOD) accepts a 3-by-3 binary
%   neighborhood, NHOOD, and returns a 1 if the center element is an
%   end point; otherwise it returns a 0.

interval1 = [0  1  0; -1  1 -1; -1 -1 -1];
interval2 = [1 -1 -1; -1  1 -1; -1 -1 -1];

% Use bwhitmiss to see if the input neighborhood matches either
% interval1 or interval2, or any of their 90-degree rotations.
for k = 1:4
   % rot90(A, k) rotates the matrix A by 90 degrees k times.
   C = bwhitmiss(nhood, rot90(interval1, k));
   D = bwhitmiss(nhood, rot90(interval2, k));
   if (C(2,2) == 1) || (D(2,2) == 1)
      % Pixel neighborhood matches one of the end-point
      % configurations, so return true.
      is_end_point = true;
      return
   end
end

% Pixel neighborhood did not match any of the end-point
% configurations, so return false.
is_end_point = false;
```

Figure 10.14 illustrates the use of function `endpoints`. Figure 10.14(a) is a binary image containing a morphological skeleton (see Section 10.3.4), and Fig. 10.14(b) shows the output of function `endpoints`.

EXAMPLE 10.6: Playing Conway's Game of Life using binary images and look-up-table-based computations.

■ An interesting and instructive application of lookup tables is the implementation of Conway's "Game of Life," which involves "organisms" arranged on a rectangular grid (see Gardner [1970, 1971]). We include it here as another illustration of the power and simplicity of lookup tables. There are simple rules for how the organisms in Conway's game are born, survive, and die from one "generation" to the next. A binary image is a convenient representation for the game, where each foreground pixel represents a living organism in that location.

Conway's genetic rules describe how to compute the next generation (next



a b

FIGURE 10.14
(a) Image of a morphological skeleton.
(b) Output of function `endpoints`. The pixels in (b) were enlarged for clarity.

binary image) from the current one:

1. Every foreground pixel with two or three neighboring foreground pixels survives to the next generation.
2. Every foreground pixel with zero, one, or at least four foreground neighbors "dies" (becomes a background pixel) because of "isolation" or "overpopulation."
3. Every background pixel adjacent to exactly three foreground neighbors is a "birth" pixel and becomes a foreground pixel.

All births and deaths occur simultaneously in the process of computing the next binary image depicting the next generation.

To implement the game of life using `makelut` and `applylut`, we first write a function that applies Conway's genetic laws to a single pixel and its 3×3 neighborhood:

`conwaylaws`

```
function out = conwaylaws(nhood)
%CONWAYLAWS Applies Conway's genetic laws to a single pixel.
%   OUT = CONWAYLAWS(NHOOD) applies Conway's genetic laws to a single
%   pixel and its 3-by-3 neighborhood, NHOOD.
num_neighbors = sum(nhood(:)) - nhood(2, 2);
if nhood(2, 2) = = 1
   if num_neighbors <= 1
```

```
        out = 0; % Pixel dies from isolation.
    elseif num_neighbors > = 4
        out = 0; % Pixel dies from overpopulation.
    else
        out = 1; % Pixel survives.
    end
else
    if num_neighbors == 3
        out = 1; % Birth pixel.
    else
        out = 0; % Pixel remains empty.
    end
end
```

See Section 2.10.4 regarding function handles.

The lookup table is constructed next by calling `makelut` with a function handle to `conwaylaws`:

```
>> lut = makelut(@conwaylaws, 3);
```

Various starting images have been devised to demonstrate the effect of Conway's laws on successive generations (see Gardner [1970, 1971]). Consider, for example, an initial image called the "Cheshire cat configuration,"

```
>> bw1 = [0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0
          0 0 0 1 0 0 1 0 0 0
          0 0 0 1 1 1 1 0 0 0
          0 0 1 0 0 0 0 1 0 0
          0 0 1 0 1 1 0 1 0 0
          0 0 1 0 0 0 0 1 0 0
          0 0 0 1 1 1 1 0 0 0
          0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0];
```

The following commands carry the computation and display up to the third generation:

The parameters `'InitialMagnification'`, `'fit'` forces the image being displayed to fit in the available display area.

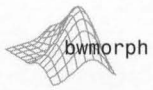
```
>> imshow(bw1, 'InitialMagnification', 'fit'), title('Generation 1')
>> bw2 = applylut(bw1, lut);
>> figure, imshow(bw2, 'InitialMagnification', 'fit'); title('Generation 2')
>> bw3 = applylut(bw2, lut);
>> figure, imshow(bw3, 'InitialMagnification', 'fit'); title('Generation 3')
```

We leave it as an exercise to show that after a few generations the cat fades to a "grin" before finally leaving a "paw print." ■

10.3.4 Function bwmorph

Toolbox function bwmorph implements a variety of morphological operations based on combinations of dilations, erosions, and lookup table operations. Its calling syntax is

```
g = bwmorph(f, operation, n)
```



where f is an input binary image, operation is a string specifying the desired operation, and n is a positive integer specifying the number of times the operation should be repeated. If argument n is omitted, the operation is performed once. Table 10.3 lists the set of valid operations for bwmorph. In the rest of this section we concentrate on two of these: *thinning* and *skeletonizing*.

Thinning means reducing binary objects or shapes in an image to strokes whose width is one pixel. For example, the fingerprint ridges shown in Fig. 10.11(c) are fairly thick. It usually is desirable for subsequent shape analysis to thin the ridges so that each is one pixel thick. Each application of thinning removes one or two pixels from the thickness of binary image objects. The following commands, for example, display the results of applying the thinning operation one and two times.

```
>> f = imread('fingerprint_cleaned.tif');
>> g1 = bwmorph(f, 'thin', 1);
>> g2 = bwmorph(f, 'thin', 2);
>> imshow(g1); figure, imshow(g2)
```

Figures 10.15(a) and 10.15(b), respectively, show the results. An important question is how many times to apply the thinning operation. For several operations, including thinning, bwmorph allows n to be set to infinity (Inf). Calling bwmorph with n = Inf instructs the function to repeat the operation until the image stops changing. This is called repeating an operation *until stability*. For example,

```
>> ginf = bwmorph(f, 'thin', Inf);
>> imshow(ginf)
```

As Fig. 10.15(c) shows, this is a significant improvement over the previous two images in terms of thinning.

Skeletonization is another way to reduce binary image objects to a set of thin strokes that retain important information about the shape of the original objects. (Skeletonization is described in more detail in Gonzalez and Woods [2008].) Function bwmorph performs skeletonization when operation is set to 'skel'. Let f denote the image of the bone-like object in Fig. 10.16(a). To compute its skeleton, we call bwmorph, with n = Inf:

```
>> fs = bwmorph(f, 'skel', Inf);
>> imshow(f); figure, imshow(fs)
```

TABLE 10.3 Operations supported by function `bwmorph`.

Operation	Description
`bothat`	"Bottom-hat" operation using a 3×3 structuring element; use `imbothat` (see Section 10.6.2) for other structuring elements.
`bridge`	Connect pixels separated by single-pixel gaps.
`clean`	Remove isolated foreground pixels.
`close`	Closing using a 3×3 structuring element of 1s; use `imclose` for other structuring elements.
`diag`	Fill in around diagonally-connected foreground pixels.
`dilate`	Dilation using a 3×3 structuring element of 1s; use `imdilate` for other structuring elements.
`erode`	Erosion using a 3×3 structuring element of 1s; use `imerode` for other structuring elements.
`fill`	Fill in single-pixel "holes" (background pixels surrounded by foreground pixels); use `imfill` (see Section 11.1.2) to fill in larger holes.
`hbreak`	Remove H-connected foreground pixels.
`majority`	Make pixel p a foreground pixel if at least five pixels in $N_8(p)$ (see Section 10.4) are foreground pixels; otherwise make p a background pixel.
`open`	Opening using a 3×3 structuring element of 1s; use function `imopen` for other structuring elements.
`remove`	Remove "interior" pixels (foreground pixels that have no background neighbors).
`shrink`	Shrink objects with no holes to points; shrink objects with holes to rings.
`skel`	Skeletonize an image.
`spur`	Remove spur pixels.
`thicken`	Thicken objects without joining disconnected 1s.
`thin`	Thin objects without holes to minimally-connected strokes; thin objects with holes to rings.
`tophat`	"Top-hat" operation using a 3×3 structuring element of 1s; use `imtophat` (see Section 10.6.2) for other structuring elements.

Figure 10.16(b) shows the resulting skeleton, which is a reasonable likeness of the basic shape of the object.

Skeletonization and thinning often produce short extraneous *spurs*, called *parasitic components*. The process of cleaning up (or removing) these spurs is called *pruning*. We can use function `endpoints` (Section 10.3.3) for this purpose. The approach is to iteratively identify and remove endpoints. The following commands, for example, post-processes the skeleton image `fs` through five iterations of endpoint removals:



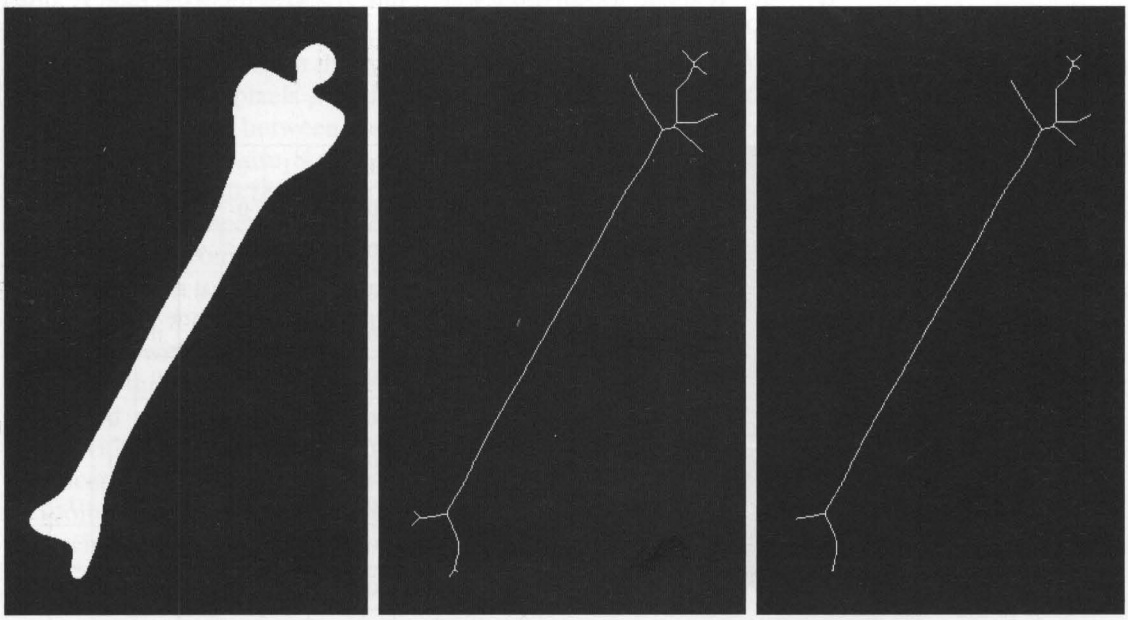
a b c

FIGURE 10.15 (a) Fingerprint image from Fig. 10.11(c) thinned once. (b) Image thinned twice. (c) Image thinned until stability.

```
>> for k = 1:5
    fs = fs & ~endpoints(fs);
end
```

Figure 10.16(c) shows the result. We would obtain similar results using the `'spur'` option from Table 10.3

```
fs = bwmorph(fs, 'spur', 5);
```



a b c

FIGURE 10.16 (a) Bone image. (b) Skeleton obtained using function `bwmorph`. (c) Resulting skeleton after pruning with function `endpoints`.

The results would not be the exactly the same because of differences in algorithm implementation. Using `Inf` instead of `5` in `bwmorph` would reduce the image to a single point.

10.4 Labeling Connected Components

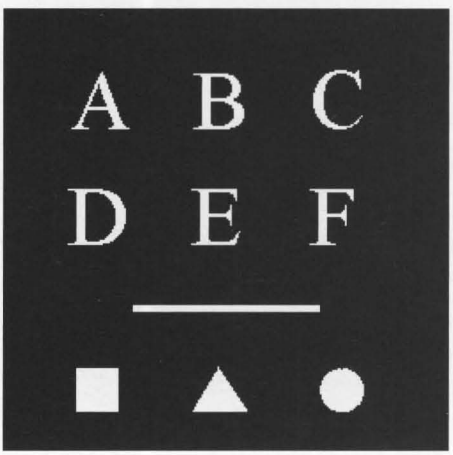
The concepts discussed thus far are applicable mostly to all foreground (or all background) individual pixels and their immediate neighbors. In this section we consider the important "middle ground" between individual foreground pixels and the set of all foreground pixels. This leads to the notion of *connected components*, also referred to as *objects* in the following discussion.

When asked to count the objects in Fig. 10.17(a), most people would identify ten: six characters and four simple geometric shapes. Figure 10.17(b) shows a small rectangular section of pixels in the image. How are the sixteen foreground pixels in Fig. 10.17(b) related to the ten objects in the image? Although they appear to be in two separate groups, all sixteen pixels actually belong to the letter "E" in Fig. 10.17(a). To develop computer programs that locate and operate on objects such as the letter "E," we need a more precise set of definitions for key terms.

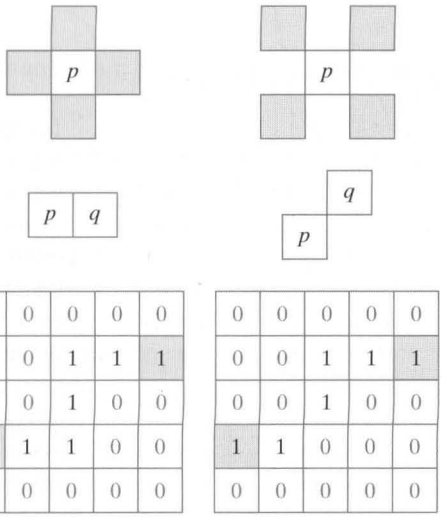
A pixel p at coordinates (x, y) has two horizontal and two vertical neighbors whose coordinates are $(x+1, y)$, $(x-1, y)$, $(x, y+1)$, and $(x, y-1)$. This set of *neighbors* of p, denoted $N_4(p)$, is shaded in Fig. 10.18(a). The four diagonal neighbors of p have coordinates $(x+1, y+1)$, $(x+1, y-1)$, $(x-1, y+1)$, and $(x-1, y-1)$. Figure 10.18(b) shows these neighbors, which are denoted $N_D(p)$. The union of $N_4(p)$ and $N_D(p)$ in Fig. 10.18(c) are the 8-*neighbors* of p, denoted $N_8(p)$.

a b

FIGURE 10.17
(a) Image containing ten objects.
(b) A subset of pixels from the image.



0	1	1	1	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	0	0	0



a b c
d e
f g

FIGURE 10.18
(a) Pixel p and its 4-neighbors, (b) Pixel p and its diagonal neighbors, (c) Pixel p and its 8-neighbors, (d) Pixels p and q are 4-adjacent and 8-adjacent. (e) Pixels p and q are 8-adjacent but not 4-adjacent. (f) The shaded pixels are both 4-connected and 8-connected. (g) The shaded pixels are 8-connected but not 4-connected.

Two pixels p and q are said to be 4-*adjacent* if $q \in N_4(p)$. Similarly, p and q are said to be 8-*adjacent* if $q \in N_8(p)$. Figures 10.18(d) and (e) illustrate these concepts. A *path* between pixels p_1 and p_n is a sequence of pixels $p_1, p_2, \ldots, p_{n-1}, p_n$ such that p_k is adjacent to p_{k+1}, for $1 \leq k < n$. A path can be 4-*connected* or 8-*connected*, depending on the type of adjacency used.

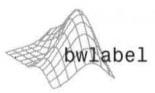
Two foreground pixels p and q are said to be 4-*connected* if there exists a 4-connected path between them, consisting entirely of foreground pixels [Fig. 10.18(f)]. They are 8-*connected* if there exists an 8-connected path between them [Fig. 10.18(g)]. For any foreground pixel, p, the set of all foreground pixels connected to it is called the *connected component* containing p.

See Section 12.1 for further discussion of connected components.

A connected component was just defined in terms of a path, and the definition of a path in turn depends on the type of adjacency used. This implies that the nature of a connected component depends on which form of adjacency we choose, with 4- and 8-adjacency being the most common. Figure 10.19 illustrates the effect that adjacency can have on determining the number of connected components in an image. Figure 10.19(a) shows a small binary image with four 4-connected components. Figure 10.19(b) shows that choosing 8-adjacency reduces the number of connected components to two.

Toolbox function `bwlabel` computes all the connected components in a binary image. The calling syntax is

```
[L, num] = bwlabel(f, conn)
```



where `f` is an input binary image and `conn` specifies the desired connectivity (either 4 or 8). Output `L` is called a *label matrix*, and `num` (optional) gives the

a b
c d

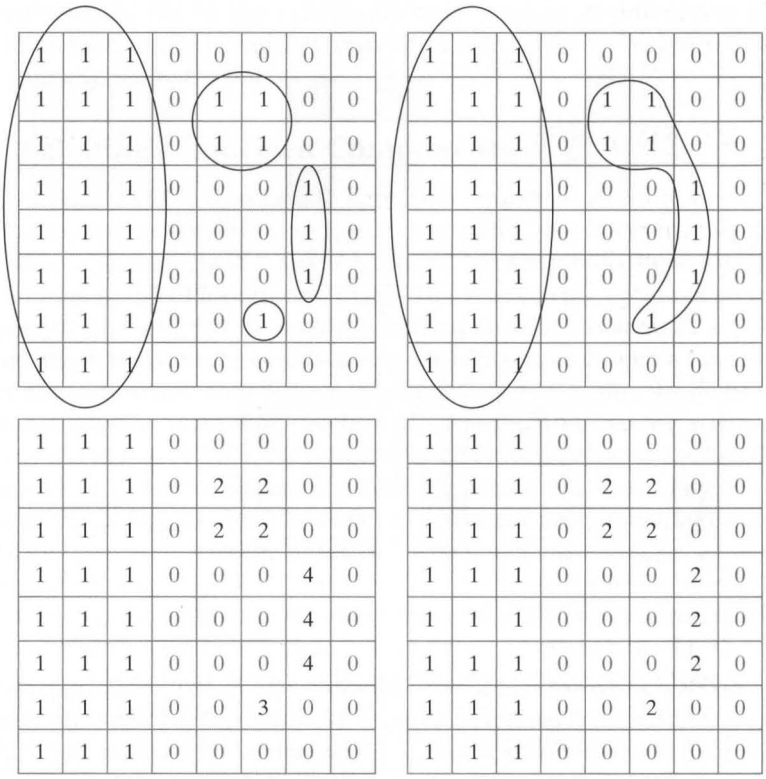


FIGURE 10.19 Connected components. (a) Four 4-connected components. (b) Two 8-connected components. (c) Label matrix obtained using 4-connectivity (d) Label matrix obtained using 8-connectivity.

total number of connected components found. If parameter `conn` is omitted, its value defaults to 8. Figure 10.19(c) shows the label matrix for the image in Fig. 10.19(a), computed using `bwlabel(f, 4)`. The pixels in each different connected component are assigned a unique integer, from 1 to the total number of connected components found. In other words, the set of pixels labeled 1 belong to the first connected component; the set of pixels labeled 2 belong to the second connected component; and so on. Background pixels are labeled 0. Figure 10.19(d) shows the label matrix corresponding to Fig. 10.19(a), computed using `bwlabel(f, 8)`.

EXAMPLE 10.7: Computing and displaying the center of mass of connected components.

■ This example shows how to compute and display the center of mass of each connected component in Fig. 10.17(a). First, we use `bwlabel` to compute the 8-connected components:

```
>> f = imread('objects.tif');
>> [L, n] = bwlabel(f);
```

Function `find` (Section 5.2.2) is useful when working with label matrices. For example, the following call to `find` returns the row and column indices for

all the pixels belonging to the third object:

```
>> [r, c] = find(L == 3);
```

Function mean with r and c as inputs then computes the center of mass of this object.

```
>> rbar = mean(r);
>> cbar = mean(c);
```

A loop can be used to compute and display the centers of mass of all the objects in the image. To make the centers of mass visible when superimposed on the image, we display them using a white " * " marker on top of a black-filled circle marker, as follows:

See Section 12.4.1 for a discussion of function regionprops, which provides a faster and more convenient way to compute object centroids.

```
>> imshow(f)
>> hold on % So later plotting commands plot on top of the image.
>> for k = 1:n
   [r, c] = find(L == k);
   rbar = mean(r);
   cbar = mean(c);
   plot(cbar, rbar, 'Marker', 'o', 'MarkerEdgeColor', 'k',...
        'MarkerFaceColor', 'k', 'MarkerSize', 10);
   plot(cbar, rbar, 'Marker', '*', 'MarkerEdgeColor', 'w');
end
```

Figure 10.20 shows the result. ■

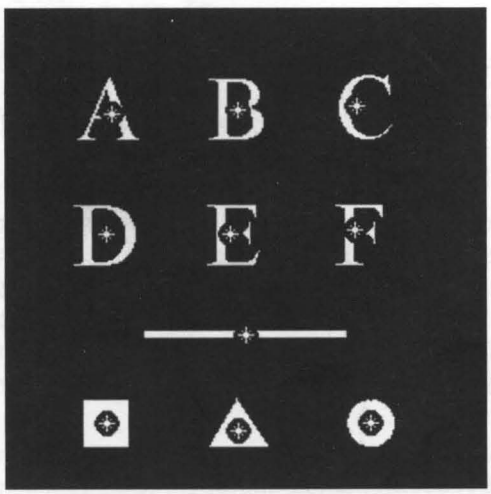


FIGURE 10.20 Centers of mass (white asterisks) shown superimposed on their corresponding connected components.

10.5 Morphological Reconstruction

See Sections 11.4.2 and 11.4.3 for additional applications of morphological reconstruction.

Reconstruction is a morphological transformation involving two images and a structuring element (instead of a single image and structuring element). One image, the *marker*, is the starting point for the transformation. The other image, the *mask*, constrains the transformation. The structuring element used defines connectivity. In this section we use 8-connectivity (the default), which implies that B in the following discussion is a 3×3 matrix of 1s, with the center defined at coordinates (2, 2). In this section we deal with binary images; gray-scale reconstruction is discussed in Section 10.6.3.

If G is the mask and F is the marker, the reconstruction of G from F, denoted $R_G(F)$, is defined by the following iterative procedure:

This definition of reconstruction is based on dilation. It is possible to define a similar operation using erosion. The results are duals of each other with respect to set complementation. These concepts are developed in detail in Gonzalez and Woods [2008].

1. Initialize h_1 to be the marker image, F.
2. Create the structuring element: `B = ones(3)`.
3. Repeat:

$$h_{k+1} = (h_k \oplus B) \cap G$$

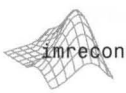
   until $h_{k+1} = h_k$.

4. $R_G(F) = h_{k+1}$.

Marker F must be a subset of G:

$$F \subseteq G$$

Figure 10.21 illustrates the preceding iterative procedure. Although this iterative formulation is useful conceptually, much faster computational algorithms exist. Toolbox function `imreconstruct` uses the "fast hybrid reconstruction" algorithm described in Vincent [1993]. The calling syntax for `imreconstruct` is




```
out = imreconstruct(marker, mask)
```

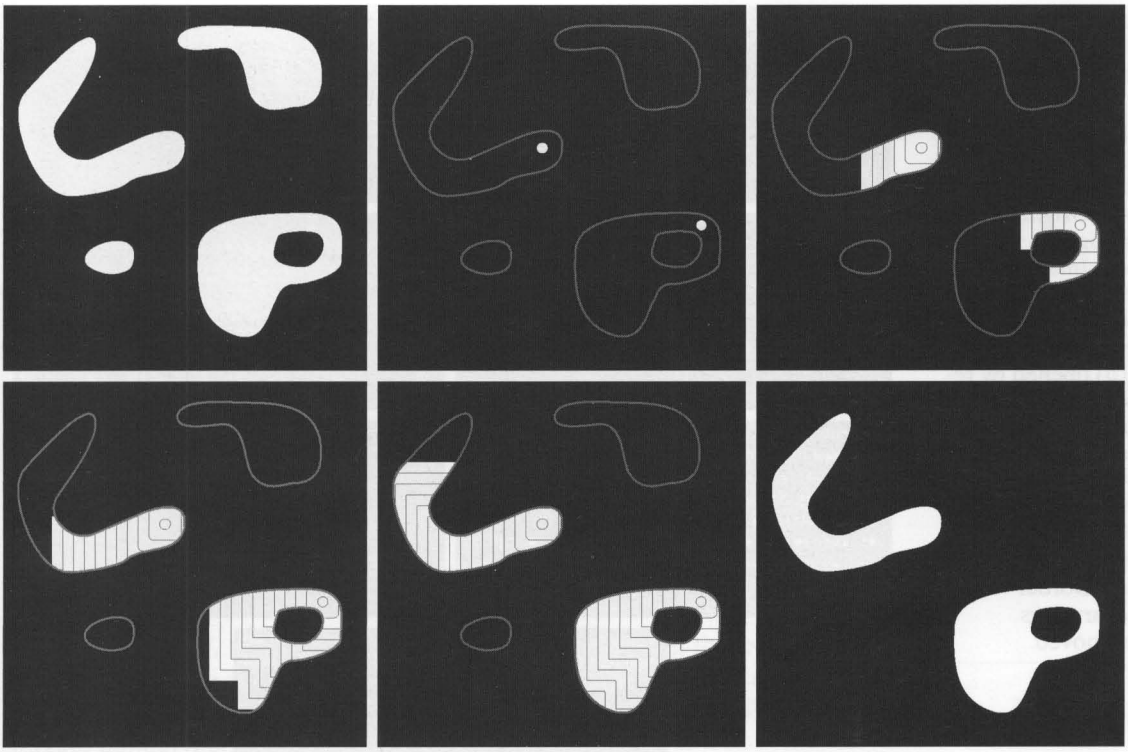
where `marker` and `mask` are as defined at the beginning of this section.

10.5.1 Opening by Reconstruction

In morphological opening, erosion typically removes small objects, and the subsequent dilation tends to restore the shape of the objects that remain. However, the accuracy of this restoration depends on the similarity between the shapes and the structuring element. The method discussed in this section, *opening by reconstruction*, restores the original shapes of the objects that remain after erosion. The opening by reconstruction of an image G using structuring element B, is defined as $R_G(G \ominus B)$.

EXAMPLE 10.8: Opening by reconstruction.

■ A comparison between opening and opening by reconstruction for an image containing text is shown in Fig. 10.22. In this example, we are interested in extracting from Fig. 10.22(a) the characters that contain long vertical strokes.



a b c
d e f

FIGURE 10.21 Morphological reconstruction. (a) Original image (the mask). (b) Marker image. (c)–(e) Intermediate result after 100, 200, and 300 iterations, respectively. (f) Final result. (The outlines of the objects in the mask image are superimposed on (b)–(e) as visual references.)

Because both opening and opening by reconstruction have erosion in common, we perform that step first, using a thin, vertical structuring element of length proportional to the height of the characters:

```
>> f = imread('book_text_bw.tif');
>> fe = imerode(f, ones(51, 1));
```

Figure 10.22(b) shows the result. The opening, shown in Fig. 10.22(c), is computed using `imopen`:

```
>> fo = imopen(f,ones(51, 1));
```

Note that the vertical strokes were restored, but not the rest of the characters containing the strokes. Finally, we obtain the reconstruction:

```
>> fobr = imreconstruct(fe, f);
```

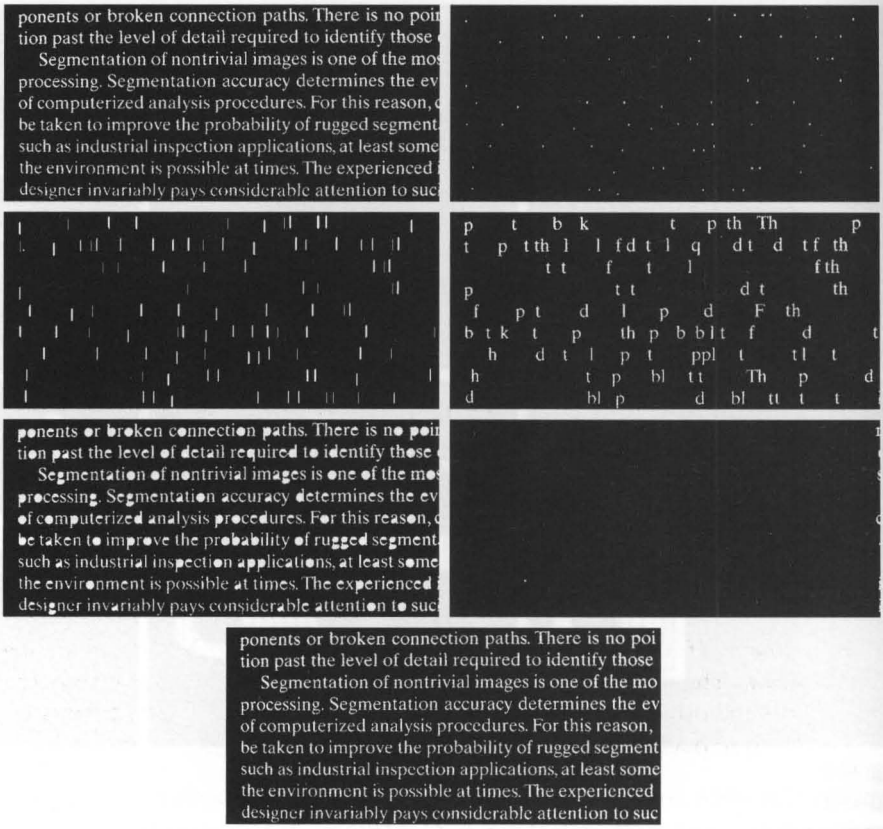


FIGURE 10.22 Morphological reconstruction: (a) Original image. (b) Image eroded with vertical line; (c) opened with a vertical line; and (d) opened by reconstruction with a vertical line. (e) Holes filled. (f) Characters touching the border (see right border). (g) Border characters removed.

The result in Fig. 10.22(d) shows that characters containing long vertical strokes were restored exactly; all other characters were removed. The remaining parts of Fig. 10.22 are explained in the following two sections. ■

10.5.2 Filling Holes

Morphological reconstruction has a broad spectrum of practical applications, each characterized by the selection of the marker and mask images. For example, let I denote a binary image and suppose that we choose the marker image, F, to be 0 everywhere except on the image border, where it is set to $1 - I$:

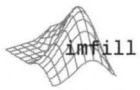
$$F(x,y) = \begin{cases} 1 - I(x,y) & \text{if } (x,y) \text{ is on the border of } I \\ 0 & \text{otherwise} \end{cases}$$

Then,

$$H = \left[R_{I^c}(F)\right]^c$$

is a binary image equal to I with all holes filled, as illustrated in Fig. 10.22(e).

Toolbox function `imfill` performs this computation automatically when the optional argument `'holes'` is used:




```
g = imfill(f, 'holes')
```

This function is discussed in more detail in Section 12.1.2.

10.5.3 Clearing Border Objects

Another useful application of reconstruction is removing objects that touch the border of an image. Again, the key task is to select the appropriate marker to achieve the desired effect. Suppose we define the marker image, F, as

$$F(x,y) = \begin{cases} I(x,y) & \text{if } (x,y) \text{ is on the border of } I \\ 0 & \text{otherwise} \end{cases}$$

where I is the original image. Then, using I as the mask image, the reconstruction

$$H = R_I(F)$$

yields an image, H, that contains only the objects touching the border, as Fig. 10.22(f) shows. The difference, $1 - H$, shown in Fig. 10.22(g), contains only the objects from the original image that do not touch the border. Toolbox function `imclearborder` performs this entire procedure automatically. Its syntax is




```
g = imclearborder(f, conn)
```

where `f` is the input image and `g` is the result. The value of `conn` can be either 4 or 8 (the default). This function suppresses structures that are lighter than their surroundings and that are connected to the image border.

10.6 Gray-Scale Morphology

All the binary morphological operations discussed in this chapter, with the exception of the hit-or-miss transform, have natural extensions to gray-scale images. In this section, as in the binary case, we start with dilation and erosion, which for gray-scale images are defined in terms of minima and maxima of pixel neighborhoods.

10.6.1 Dilation and Erosion

The *gray-scale dilation* of a gray-scale image f by structuring element b, denoted by $f \oplus b$, is defined as

$$(f \oplus b)(x,y) = \max\{f(x-x', y-y') + b(x', y') \mid (x', y') \in D_b\}$$

where D_b is the domain of b, and $f(x,y)$ is assumed to equal $-\infty$ outside the domain of f. This equation implements a process similar to spatial convolution,

explained in Section 3.4.1. Conceptually, we can think of rotating the structuring element by 180° about its origin and translating it to all locations in the image, just as a convolution kernel is rotated and then translated about the image. At each translated location, the rotated structuring element values are added to the image pixel values and the maximum is computed.

One important difference between convolution and gray-scale dilation is that, in the latter, D_b, is a binary matrix that defines which locations in the neighborhood are included in the max operation. In other words, for an arbitrary pair of coordinates (x_0, y_0) in the domain D_b, the term $f(x - x_0, y - y_0) + b(x_0, y_0)$ is included in the max computation *only if* D_b is 1 at those coordinates. This is repeated for all coordinates $(x', y') \in D_b$ each time that coordinates (x, y) change. Plotting $b(x', y')$ as a function of coordinates x' and y' would look like a digital "surface" with the height at any pair of coordinates being given by the value of b at those coordinates.

Gray-scale dilation usually is performed using *flat* structuring elements in which the value (height) of b is 0 at all coordinates over which D_b is defined. That is,

$$b(x', y') = 0 \quad \text{for } (x', y') \in D_b$$

In this case, the max operation is specified completely by the pattern of 0s and 1s in binary matrix D_b, and the gray-scale dilation equation simplifies to

$$(f \oplus b)(x, y) = \max\{f(x - x', y - y') \mid (x', y') \in D_b\}$$

Thus, flat gray-scale dilation is a local-maximum operator, where the maximum is taken over a set of pixel neighbors determined by the spatial shape of the 1-valued elements in D_b.

Nonflat structuring elements are created with function `strel` by passing it two matrices: (1) a matrix of 0s and 1s specifying the structuring element domain, and (2) a second matrix specifying height values, For example,

```
>> b = strel([1 1 1], [1 2 1])
    b =
    Nonflat STREL object containing 3 neighbors.
    Neighborhood:
      1     1     1
Height:
      1     2     1
```

creates a 1×3 structuring element whose height values are $b(0, -1) = 1$, $b(0, 0) = 2$, and $b(0, 1) = 1$.

Flat structuring elements for gray-scale images are created using `strel` in the same way as for binary images. For example, the following commands show how to dilate the image `f` in Fig. 10.23(a) using a flat 3×3 structuring element:



a b
c d

FIGURE 10.23 Dilation and erosion. (a) Original image. (b) Dilated image. (c) Eroded image. (d) Morphological gradient. (Original image courtesy of NASA.)

```
>> se = strel('square', 3);
>> gd = imdilate(f, se);
```

Figure 10.23(b) shows the result. As expected, the image is slightly blurred. The rest of this figure is explained in the following discussion.

The *gray-scale erosion* of gray-scale image f by structuring element b, denoted by $f \ominus b$, is defined as

$$(f \ominus b)(x, y) = \min\{f(x + x', y + y') - b(x', y') \mid (x', y') \in D_b\}$$

where D_b is the domain of b and f is assumed to be $+\infty$ outside the domain of f. As before, we think geometrically in terms of translating the structuring element to all locations in the image. At each translated location, the structuring element values are subtracted from the image pixel values and the minimum is computed.

As with dilation, gray-scale erosion usually is performed using flat structuring elements. The equation for flat gray-scale erosion then simplifies to

$$(f \ominus b)(x, y) = \min\{f(x + x', y + y') \mid (x', y') \in D_b\}$$

Thus, flat gray-scale erosion is a local-minimum operator, in which the minimum is taken over a set of pixel neighbors determined by the spatial shape of the 1-valued elements of D_b. Figure 10.23(c) shows the result of using function `imerode` with the same structuring element that was used for Fig. 10.23(b):

```
>> ge = imerode(f, se);
```

Dilation and erosion can be combined to achieve a variety of effects. For example, subtracting an eroded image from its dilated version produces a "morphological gradient," which is a measure of local gray-level variation in the image. For example, letting

```
>> morph_grad = gd - ge;
```

produced the image in Fig. 10.23(d), which is the morphological gradient of the image in Fig. 10.23(a). This image has edge-enhancement characteristics similar to those that would be obtained using the gradient operations discussed in Sections 7.6.1 and later in Section 11.1.3.

10.6.2 Opening and Closing

The expressions for opening and closing gray-scale images have the same form as their binary counterparts. The *opening* of gray-scale image f by structuring element b, denoted $f \circ b$, is defined as

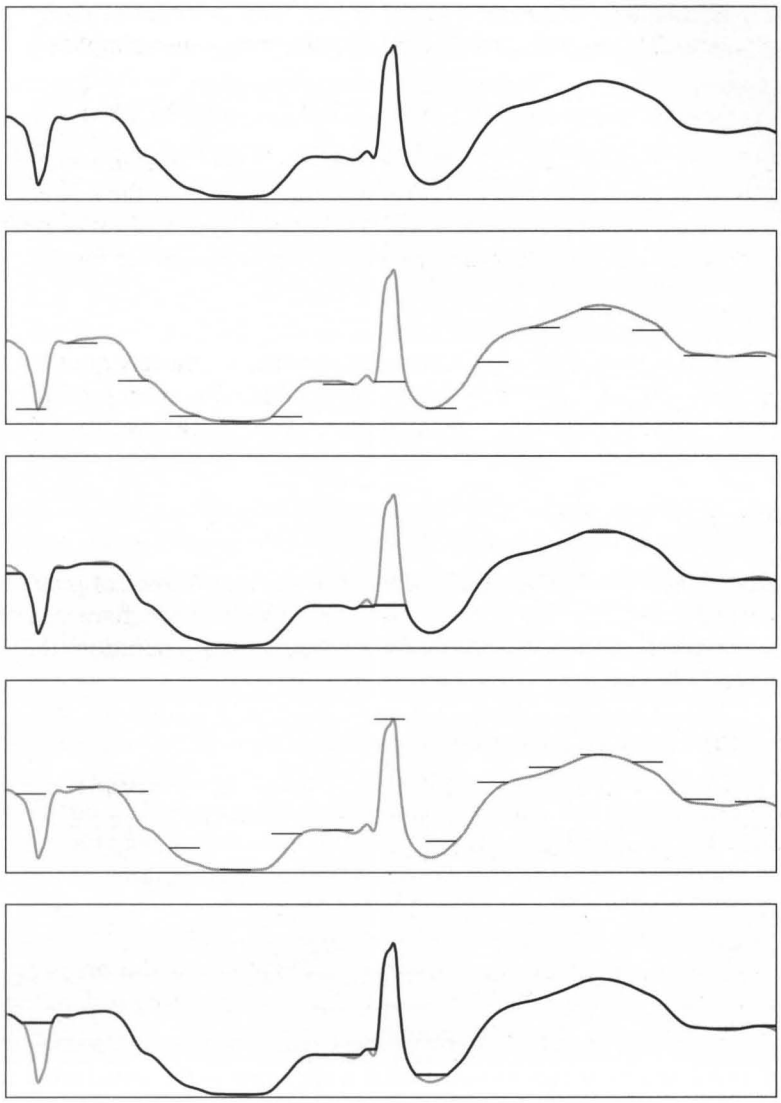
$$f \circ b = (f \ominus b) \oplus b$$

where it is understood that erosion and dilation are the grayscale operations defined in Section 10.6.1. Similarly, the *closing* of f by b, denoted $f \bullet b$, is defined as dilation followed by erosion:

$$f \bullet b = (f \oplus b) \ominus b$$

Both operations have simple geometric interpretations. Suppose that an image function $f(x, y)$ is viewed as a 3-D surface; that is, its intensity values are interpreted as height values over the xy-plane. Then the opening of f by b can be interpreted geometrically as pushing structuring element b up against the underside of the surface and translating it across the entire domain of f. The opening is constructed by finding the highest points reached by any part of the structuring element as it slides against the undersurface of f.

Figure 10.24 illustrates the concept in one dimension. Consider the curve in Fig. 10.24(a) to be the values along a single row of an image. Figure 10.24(b) shows a flat structuring element in several positions, pushed up against the underside of the curve. The complete opening is shown as the heavy curve



a
b
c
d
e

FIGURE 10.24 Opening and closing in one dimension. (a) Original 1-D signal. (b) Flat structuring element pushed up underneath the signal. (c) Opening. (d) Flat structuring element pushed down along the top of the signal. (e) Closing.

in Fig. 10.24(c). Because the structuring element is too large to fit inside the upward peak on the middle of the curve, that peak is removed by the opening. In general, openings are used to remove small bright details while leaving the overall gray levels and larger bright features relatively undisturbed.

Figure 10.24(d) is a graphical illustration of closing. The structuring element is pushed down on top of the curve while being translated to all locations. The closing, shown in Fig. 10.24(e), is constructed by finding the lowest points reached by any part of the structuring element as it slides against the upper side of the curve. You can see that closing suppresses dark details smaller than the structuring element.

EXAMPLE 10.9: Morphological smoothing using openings and closings.

■ Because opening suppresses bright details smaller than the structuring element, and closing suppresses dark details smaller than the structuring element, they are used often in combination for image smoothing and noise removal. In this example we use `imopen` and `imclose` to smooth the image of wood dowel plugs shown in Fig. 10.25(a). The key feature of these dowels is their wood grain (appearing as dark streaks) superimposed on a reasonably uniform, light background. When interpreting the results that follow, it helps to keep in mind the analogies of opening and closing illustrated in Fig. 10.24.

Consider the following sequence of steps:

```
>> f = imread('plugs.jpg');
>> se = strel('disk', 5);
>> fo = imopen(f, se);
>> foc = imclose(fo, se);
```

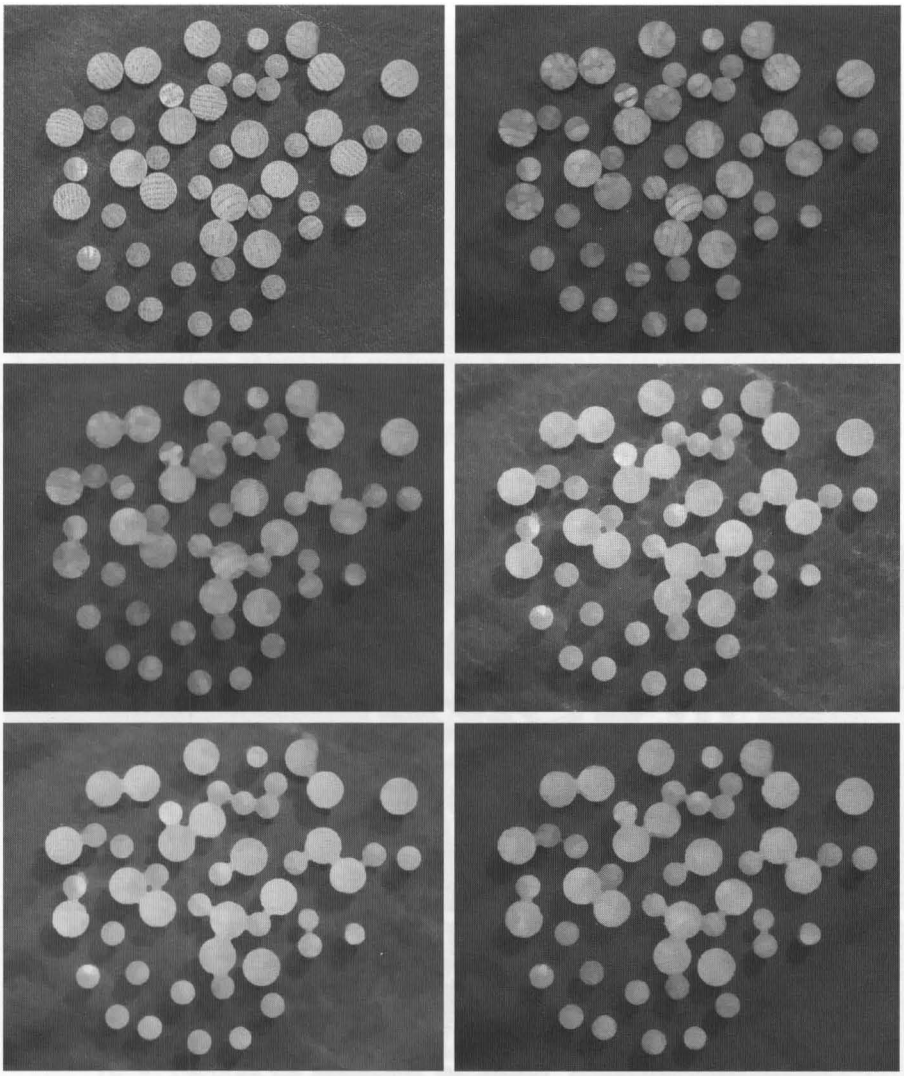
Figure 10.25(b) shows the opened image, `fo`. Here, we see that the light areas have been toned down (smoothed) and the dark streaks in the dowels have not been nearly as affected. Figure 10.25(c) shows the closing of the opening, `foc`. Now we notice that the dark areas have been smoothed as well, resulting is an overall smoothing of the entire image. This procedure is often called *open-close filtering.*

A similar procedure, called *close-open filtering,* reverses the order of the operations. Figure 10.25(d) shows the result of closing the original image. The dark streaks in the dowels have been smoothed out, leaving mostly light detail (for example, note the light streaks in the background). The opening of Fig. 10.25(d) [Fig. 10.25(e)] shows a smoothing of these streaks and further smoothing of the dowel surfaces. The net result is overall smoothing of the original image.

Another way to use openings and closings in combination is in *alternating sequential filtering*. One form of alternating sequential filtering is to perform open-close filtering with a series of structuring elements of increasing size. The following commands illustrate this process, which begins with a small structuring element and increases its size until it is the same as the structuring element used to obtain Figs. 10.25(b) and (c):

```
>> fasf = f;
>> for k = 2:5
      se = strel('disk' ,k);
      fasf = imclose(imopen(fasf, se), se);
   end
```

The result, shown in Fig. 10.25(f), yielded a slightly smoother image than using a single open-close filter, at the expense of additional processing. When comparing the three approaches in this particular case, close-open filtering yielded the smoothest result. ■



a b
c d
e f

FIGURE 10.25 Smoothing using openings and closings. (a) Original image of wood dowel plugs. (b) Image opened using a disk of radius 5. (c) Closing of the opening. (d) Closing of the original image. (e) Opening of the closing. (f) Result of alternating sequential filter.

■ Openings can be used to compensate for nonuniform background illumination. Figure 10.26(a) shows an image, `f`, of rice grains in which the background is darker towards the bottom than in the upper portion of the image. The uneven illumination makes image thresholding (Section 11.3) difficult. Figure 10.26(b), for example, is a thresholded version in which grains at the top of the image are well separated from the background, but grains at the bottom are extracted improperly from the background. Opening the image can produce a reasonable estimate of the background across the image, as long as the structuring element is large enough so that it does not fit entirely within the rice grains. For example, the commands

EXAMPLE 10.10: Compensating for a nonuniform background.

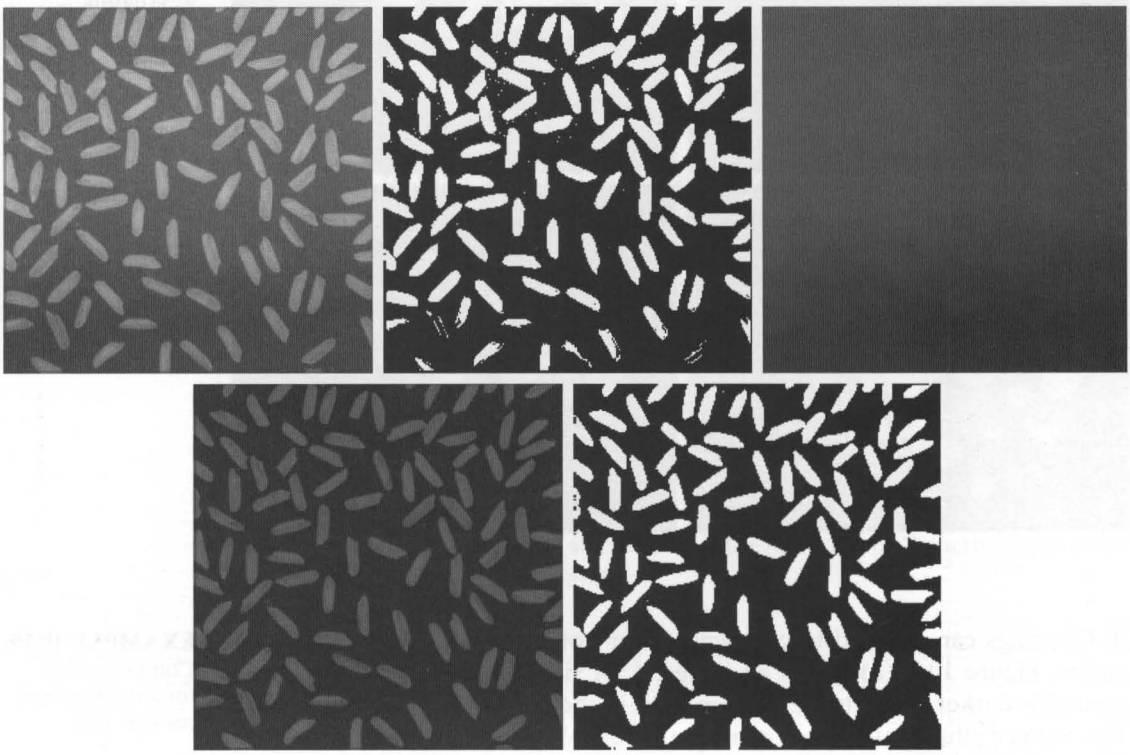
```
>> se = strel('disk', 10);
>> fo = imopen(f, se);
```

resulted in the opened image in Fig. 10.26(c). By subtracting this image from the original, we can generate an image of the grains with a reasonably uniform background:

```
>> f2 = f - fo;
```

Figure 10.26(d) shows the result, and Fig. 10.26(e) shows the new thresholded image. Note the improvement over Fig. 10.26(b). ■

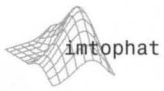
Subtracting an opened image from the original is called a *tophat* transformation. Toolbox function `imtophat` performs this operation in a single step:



a b c
d e

FIGURE 10.26 Compensating for non-uniform illumination. (a) Original image. (b) Thresholded image. (c) Opened image showing an estimate of the background. (d) Result of subtracting the estimated background for the original image. (e) Result of thresholding the image in (d). (Original image courtesy of The MathWorks, Inc.)

```
>> f2 = imtophat(f, se);
```

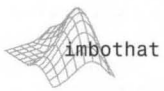


In addition to this syntax, function `imtophat` can be called as

```
g = imtophat(f, NHOOD)
```

where `NHOOD` is an array of 0s and 1s that specifies the size and shape of the structuring element. This syntax is the same as using the call

```
imtophat(f, strel(NHOOD))
```



A related function, `imbothat`, performs a *bottomhat* transformation, defined as the closing of the image minus the image. Its syntax is the same as for function `imtophat`. These two functions can be used for contrast enhancement using commands such as

```
>> se = strel('disk', 3);
>> g = f + imtophat(f, se) - imbothat(f, se);
```

EXAMPLE 10.11: Granulometry.

■ Determining the size distribution of particles in an image is an important application in the field of *granulometry*. Morphological techniques can be used to measure particle size distribution indirectly; that is, without having to identify and measure each particle explicitly. For particles with regular shapes that are lighter than the background, the basic approach is to apply morphological openings of increasing size. For each opening, the sum of all the pixel values in the opening is computed; this sum sometimes is called the *surface area* of the image. The following commands apply disk-shaped openings with radii 0 to 35 to the image in Fig. 10.25(a):

```
>> f = imread('plugs.jpg');
>> sumpixels = zeros(1, 36);
>> for k = 0:35
       se = strel('disk', k);
       fo = imopen(f, se);
       sumpixels(k + 1) = sum(fo(:));
end

>> plot(0:35, sumpixels), xlabel('k'), ylabel('Surface area')
```

If `v` is a vector, then `diff(v)` returns a vector, one element shorter than `v`, of differences between adjacent elements. If `X` is a matrix, then `diff(X)` returns a matrix of row differences: `[X(2:end, :) - X(1:end-1, :)]`.

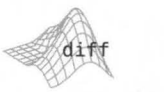


Figure 10.27(a) shows the resulting plot of `sumpixels` versus `k`. More interesting is the reduction in surface area between successive openings:

```
>> plot(-diff(sumpixels))
>> xlabel('k')
>> ylabel('Surface area reduction')
```

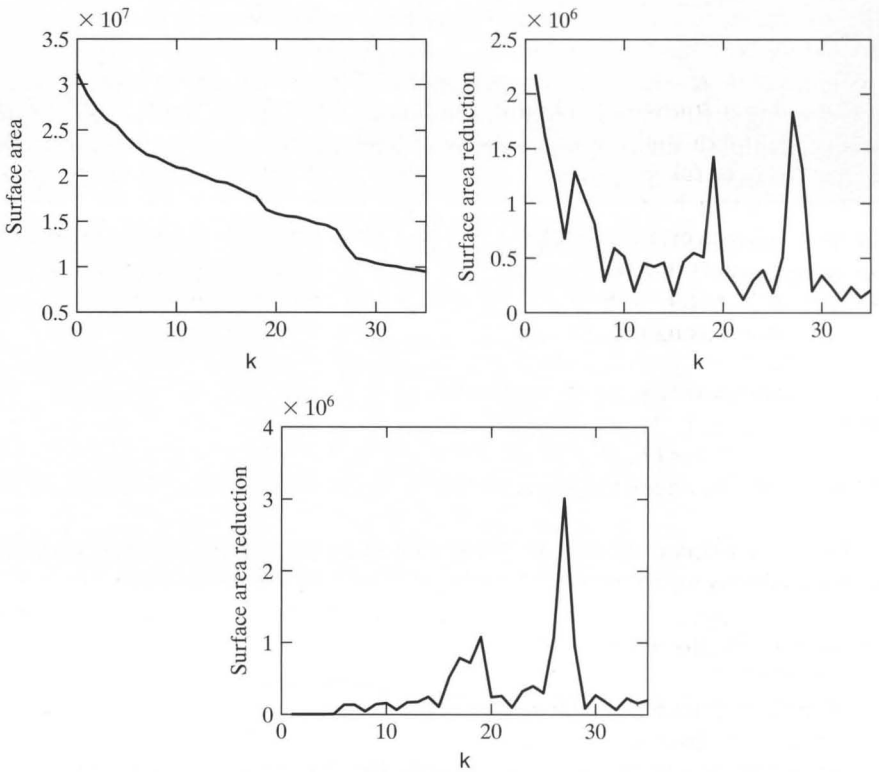
Peaks in the plot in Fig. 10.27(b) indicate the presence of a large number of

objects having that radius. Because the plot is quite noisy, we repeat this procedure with the smoothed version of the plugs image in Fig. 10.25(d). The result, shown in Fig. 10.27(c), indicates more clearly the two different sizes of objects in the original image. ■

10.6.3 Reconstruction

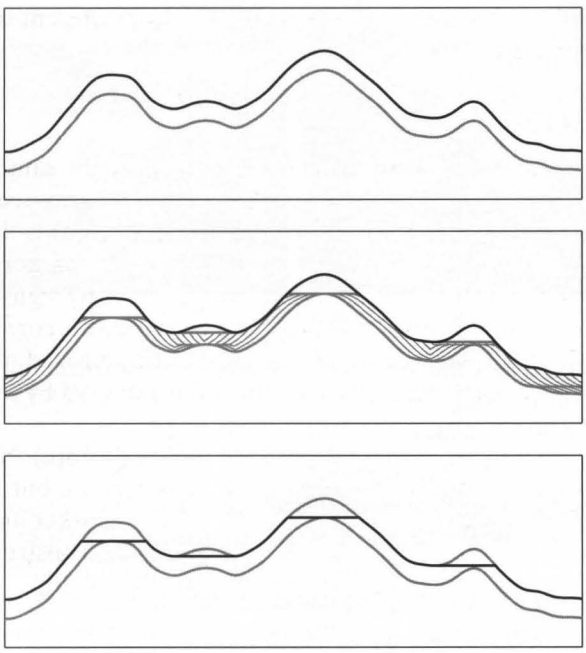
Gray-scale morphological reconstruction is defined by the same iterative procedure given in Section 10.5. Figure 10.28 shows how gray-scale reconstruction works in one dimension. The top curve of Fig. 10.28(a) is the mask while the bottom, gray curve is the marker. In this case the marker is formed by subtracting a constant from the mask, but in general any signal can be used for the marker as long as none of its values exceed the corresponding value in the mask. Each iteration of the reconstruction procedure spreads the peaks in the marker curve until they are forced downward by the mask curve [Fig. 10.28(b)].

The final reconstruction is the black curve in Fig. 10.28(c). Notice that the two smaller peaks were eliminated in the reconstruction, but the two taller peaks, although they are now shorter, remain. When a marker image is formed by subtracting a constant h from the mask image, the reconstruction is called



a b
c

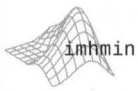
FIGURE 10.27 Granulometry. (a) Surface area versus structuring element radius. (b) Reduction in surface area versus radius. (c) Reduction in surface area versus radius for a smoothed image.



a
b
c

FIGURE 10.28 Gray-scale morphological reconstruction in one dimension. (a) Mask (top) and marker curves. (b) Iterative computation of the reconstruction. (c) Reconstruction result (black curve).

the *h-minima transform*. The h-minima transform is computed by toolbox function `imhmin` and is used to suppress small peaks.



Another useful gray-scale reconstruction technique is *opening-by-reconstruction*, in which an image is first eroded, just as in standard morphological opening. However, instead of following the opening by a closing, the eroded image is used as the marker image in a reconstruction. The original image is used as the mask. Figure 10.29(a) shows an example of opening-by-reconstruction, obtained using the commands

```
>> f = imread('plugs.jpg');
>> se = strel('disk', 5);
>> fe = imerode(f, se);
>> fobr = imreconstruct(fe, f);
```

Reconstruction can be used to clean up the image further by applying to it a *closing-by-reconstruction*. This technique is implemented by complementing an image, computing its opening-by-reconstruction, and then complementing the result, as follows:

```
>> fobrc = imcomplement(fobr);
>> fobrce = imerode(fobrc, se);
>> fobrcbr = imcomplement(imreconstruct(fobrce, fobrc));
```

a b

FIGURE 10.29
(a) Opening-by-reconstruction. (b) Opening-by-reconstruction followed by closing-by-reconstruction.

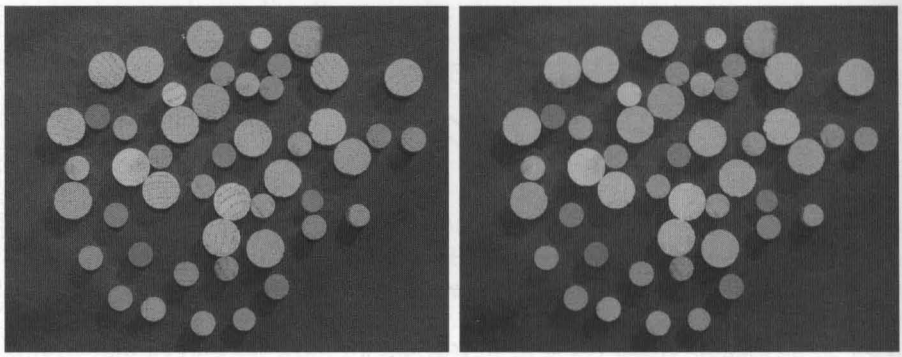


Figure 10.29(b) shows the result of opening-by-reconstruction followed by closing-by-reconstruction. Compare it with the open-close filter and alternating sequential filter results in Fig. 10.25.

EXAMPLE 10.12: Using gray-scale reconstruction to remove a complex background.

■ Our concluding example uses gray-scale reconstruction in several steps. The objective is to isolate the text out of the image of calculator keys shown in Fig. 10.30(a). The first step is to suppress the horizontal reflections on the top of each key. To accomplish this, we use the fact that these reflections are wider than any single text character in the image. We perform opening-by-reconstruction using a structuring element that is a long horizontal line:

```
>> f = imread('calculator.jpg');
>> f_obr = imreconstruct(imerode(f, ones(1, 71)), f);
>> f_o = imopen(f, ones(1, 71));  % For comparison.
```

The opening-by-reconstruction (`f_obr`) is shown in Fig. 10.30(b). For comparison, Fig. 10.30(c) shows the standard opening (`f_o`). Opening-by-reconstruction did a better job of extracting the background between horizontally adjacent keys. Subtracting the opening-by-reconstruction from the original image is called *tophat-by-reconstruction*, and is shown in Fig. 10.30(d):

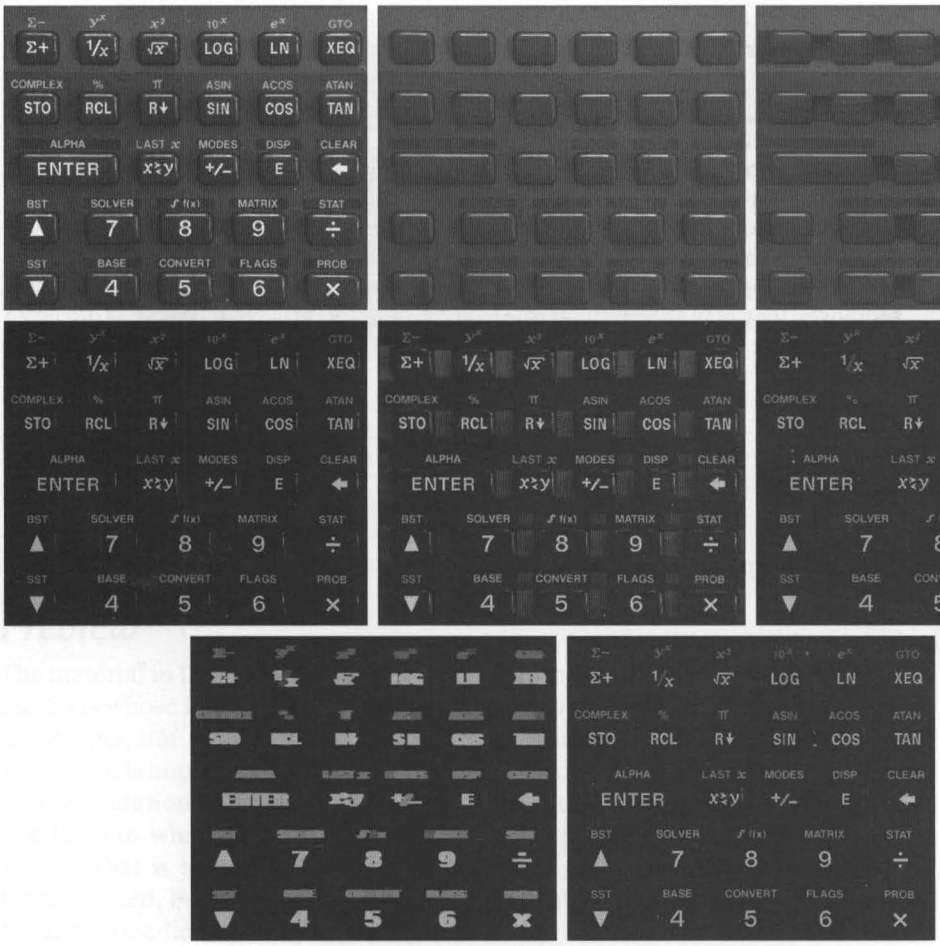
```
>> f_thr = f - f_obr;
>> f_th = f - f_o;          % Or imtophat(f,ones(1, 71))
```

Figure 10.30(e) shows the standard tophat computation (i.e., `f_th`).

Next, we suppress the vertical reflections on the right edges of the keys in Fig. 10.30(d). This is done by performing opening-by-reconstruction with a small horizontal line:

```
>> g_obr = imreconstruct(imerode(f_thr, ones(1, 11)), f_thr);
```

In the result [Fig. 10.30(f)], the vertical reflections are gone, but so are thin, vertical-stroke characters, such as the slash on the percent symbol and the "I" in ASIN. We make use of the fact that the characters that have been sup-



a b c
d e f
g h

FIGURE 10.30 An application of gray-scale reconstruction. (a) Original image. (b) Opening-by-reconstruction. (c) Opening. (d) Tophat-by-reconstruction. (e) Tophat. (f) Opening-by-reconstruction of (d) using a horizontal line. (g) Dilation of (f) using a horizontal line. (h) Final reconstruction result.

pressed in error are very close spatially to other characters still present by first performing a dilation [Fig. 10.30(g)],

```
>> g_obrd = imdilate(g_obr, ones(1, 21));
```

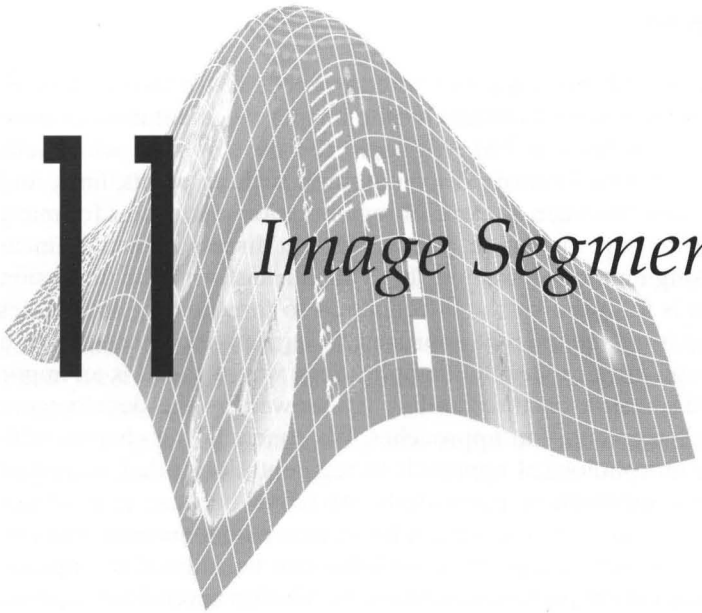
followed by a final reconstruction with `f_thr` as the mask and `min(g_obrd, f_thr)` as the marker:

```
>> f2 = imreconstruct(min(g_obrd, f_thr), f_thr);
```

Figure 10.30(h) shows the final result. Note that the shading and reflections on the background and keys were removed successfully. ■

Summary

The morphological concepts and techniques introduced in this chapter constitute a powerful set of tools for extracting features from an image. The basic operators of erosion, dilation, and reconstruction—defined for both binary and gray-scale image processing—can be used in combination to perform a wide variety of tasks. As shown in the following chapter, morphological techniques can be used for image segmentation. Moreover, they play an important role in algorithms for image description, as discussed in Chapter 12.



11 Image Segmentation

Preview

The material in the previous chapter began a transition from image processing methods whose inputs and outputs are images, to methods in which the inputs are images, but the outputs are attributes extracted from those images. Segmentation is another major step in that direction.

Segmentation subdivides an image into its constituent regions or objects. The level to which the subdivision is carried depends on the problem being solved. That is, segmentation should stop when the objects of interest have been isolated. For example, in the automated inspection of electronic assemblies, interest lies in analyzing images of the products with the objective of determining the presence or absence of specific anomalies, such as missing components or broken connection paths. There is no reason to carry segmentation past the level of detail required to identify those elements.

Segmentation of nontrivial images is one of the most difficult tasks in image processing. Segmentation accuracy determines the eventual success or failure of computerized analysis procedures. For this reason, considerable care should be taken to improve the probability of rugged segmentation. In some situations, such as industrial inspection applications, at least some measure of control over the environment is possible at times. In others, as in remote sensing, user control over image acquisition is limited principally to the choice of imaging sensors.

Segmentation algorithms for monochrome images generally are based on one of two basic properties of image intensity values: discontinuity and similarity. In the first category, the approach is to partition an image based on abrupt changes in intensity, such as edges. The principal approaches in the second category are based on partitioning an image into regions that are similar according to a set of predefined criteria.

In this chapter we discuss a number of approaches in the two categories just mentioned, as they apply to monochrome images (segmentation of color images is discussed in Section 7.6). We begin the development with methods suitable for detecting intensity discontinuities, such as points, lines, and edges. Edge detection has been a staple of segmentation algorithms for many years. In addition to edge detection per se, we also discuss detecting linear edge segments using methods based on the Hough transform. The discussion of edge detection is followed by the introduction to thresholding techniques. Thresholding also is a fundamental approach to segmentation that enjoys a high degree of popularity, especially in applications where speed is an important factor. The discussion on thresholding is followed by the development of region-oriented segmentation approaches. We conclude the chapter with a discussion of a morphological approach to segmentation called *watershed segmentation*. This approach is particularly attractive because it produces closed, well-defined region boundaries, behaves in a global manner, and provides a framework in which a priori knowledge can be utilized to improve segmentation results. As in previous chapters, we develop several new custom functions that complement the Image Processing Toolbox.

11.1 Point, Line, and Edge Detection

In this section we discuss techniques for detecting the three basic types of intensity discontinuities in a digital image: points, lines, and edges. The most common way to look for discontinuities is to run a mask through the image in the manner described in Sections 3.4 and 3.5. For a 3×3 mask this involves computing the sum of products of the coefficients with the intensity levels contained in the region encompassed by the mask. The response, R, of the mask at any point in the image is given by

$$\begin{aligned} R &= w_1 z_1 + w_2 z_2 + \ldots + w_9 z_9 \\ &= \sum_{i=1}^{9} w_i z_i \end{aligned}$$

where z_i is the intensity of the pixel associated with mask coefficient w_i. As before, the response of the mask is defined at its center.

11.1.1 Point Detection

The detection of isolated points embedded in areas of constant or nearly constant intensity in an image is straightforward in principle. Using the mask shown in Fig. 11.1, we say that an isolated point has been detected at the location on which the mask is centered if

$$|R| \geq T$$

where T is a nonnegative threshold. This approach to point detection is implemented in the toolbox using function `imfilter` with the mask in Fig. 11.1. The

−1	−1	−1
−1	8	−1
−1	−1	−1

FIGURE 11.1
A mask for point detection.

important requirements are that the strongest response of a mask be when the mask is centered on an isolated point, and that the response be 0 in areas of constant intensity.

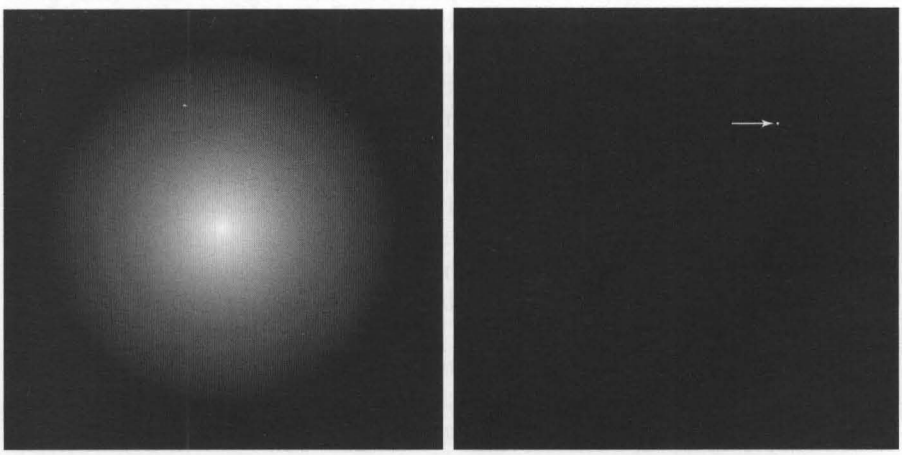
If `T` is given, the following command implements the point-detection approach just discussed:

```
>> g = abs(imfilter(tofloat(f), w)) >= T;
```

where `f` is the input image, `w` is an appropriate point-detection mask [e.g., the mask in Fig. 11.1], and `g` is an image containing the points detected. Recall from Section 3.4.1 that `imfilter` converts its output to the class of the input, so we use `tofloat(f)` in the filtering operation to prevent premature truncation of values if the input is of an integer class, and because the `abs` operation does not accept integer data. The output image `g` is of class `logical`; its values are 0 and 1. If `T` is not given, its value often is chosen based on the filtered result, in which case the previous command string is divided into three basic steps: (1) Compute the filtered image, `abs(imfilter(tofloat(f), w))`, (2) find the value for `T` using the data from the filtered image, and (3) compare the filtered image against `T`. The following example illustrates this approach.

EXAMPLE 11.1: Point detection.

■ Figure 11.2(a) shows an image, `f`, with a nearly invisible black point in the northeast quadrant of the sphere. We detect the point as follows:



a b

FIGURE 11.2
(a) Gray-scale image with a nearly invisible isolated black point in the northeast quadrant of the sphere. (b) Image showing the detected point. (The point was enlarged to make it easier to see.)

```
>> w = [-1 -1 -1; -1 8 -1; -1 -1 -1];
>> g = abs(imfilter(tofloat(f), w));
>> T = max(g(:));
>> g = g >= T;
>> imshow(g)
```

By selecting `T` to be the maximum value in the filtered image, `g`, and then finding all points in `g` such that `g >= T`, we identify the points that give the largest response. The assumption is that these are isolated points embedded in a constant or nearly constant background. Because `T` was selected in this case to be the maximum value in `g`, there can be no points in `g` with values greater than `T`; we used the `>=` operator (instead of `=`)for consistency in notation. As Fig. 11.2(b) shows, there was a single isolated point that satisfied the condition `g >= T` with `T` set to `max(g(:))`. ■

Another approach to point detection is to find the points in all neighborhoods of size $m \times n$ for which the difference of the maximum and minimum pixels values exceeds a specified value of `T`. This approach can be implemented using function `ordfilt2` introduced in Section 3.5.2:

```
>> g = ordfilt2(f, m*n, ones(m, n)) - ordfilt2(f, 1, ones(m, n));
>> g = g >= T;
```

It is easily verified that choosing $m = n = 5$ and `T = max(g(:))` yields the same result as in Fig. 11.2(b). The preceding formulation is more flexible than using the mask in Fig. 11.1. For example, if we wanted to compute the difference between the highest and the next highest pixel value in a neighborhood, we would replace the `1` on the far right of the preceding expression by `m*n - 1`. Other variations of this basic theme are formulated in a similar manner.

11.1.2 Line Detection

The next level of complexity is line detection. If the mask in Fig 11.3(a) were moved around an image, it would respond more strongly to lines (one pixel thick) oriented horizontally. With a constant background, the maximum response results when the line passes through the middle row of the mask. Similarly, the second mask in Fig. 11.3 responds best to lines oriented at +45°; the third mask to vertical lines; and the fourth mask to lines in the −45° direction.

Recall that in our image coordinate system (Fig. 2.1) the x-axis points down. Positive angles are measured counter-clockwise with respect to that axis.

a b c d

FIGURE 11.3 Line detector masks.

−1	−1	−1
2	2	2
−1	−1	−1

Horizontal

2	−1	−1
−1	2	−1
−1	−1	2

+45°

−1	2	−1
−1	2	−1
−1	2	−1

Vertical

−1	−1	2
−1	2	−1
2	−1	−1

−45°

Note that the preferred direction of each mask is weighted with a larger coefficient than other possible directions. The coefficients of each mask sum to zero, indicating a zero response in areas of constant intensity.

Let R_1, R_2, R_3, and R_4 denote the responses of the masks in Fig. 11.3, from left to right, where the R's are given by the equation in the previous section. Suppose that the four masks are run individually through an image. If, at a certain point in the image, $|R_i| > |R_j|$ for all $j \neq i$, that point is said to be more likely associated with a line in the direction favored by mask i. If we are interested in detecting all the lines in an image in the direction defined by a given mask, we simply run the mask through the image and threshold the absolute value of the result. The points that are left are the strongest responses, which, for lines one pixel thick, correspond closest to the direction defined by the mask. The following example illustrates this procedure.

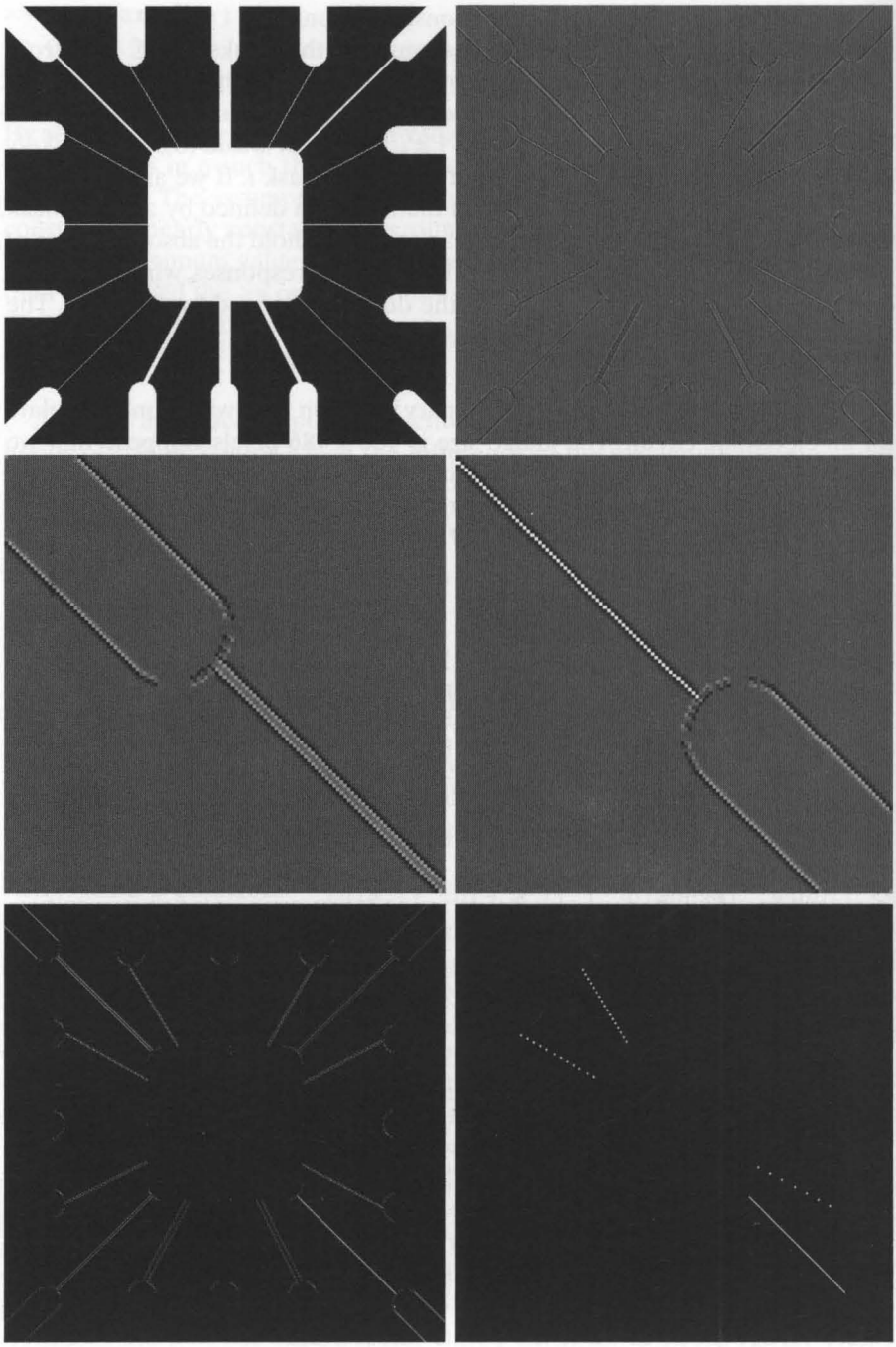
EXAMPLE 11.2: Detecting lines in a specified direction.

■ Figure 11.4(a) shows a digitized (binary) portion of a wire-bond template for an electronic circuit. The image size is 486 × 486 pixels. Suppose that we want to find all the lines that are one pixel thick, oriented at +45°. For this, we use the second mask in Fig. 11.3. Figures 11.4(b) through (f) were generated using the following commands, where `f` is the image in Fig. 11.4(a):

```
>> w = [2 -1 -1; -1 2 -1; -1 -1 2];
>> g = imfilter(tofloat(f), w);
>> imshow(g, [ ]) % Fig. 11.4(b)
>> gtop = g(1:120, 1:120); % Top, left section.
>> gtop = pixeldup(gtop, 4); % Enlarge by pixel duplication.
>> figure, imshow(gtop, [ ]) % Fig. 11.4(c)
>> gbot = g(end - 119:end, end - 119:end);
>> gbot = pixeldup(gbot, 4);
>> figure, imshow(gbot, [ ]) % Fig. 11.4(d)
>> g = abs(g);
>> figure, imshow(g, [ ]) % Fig. 11.4(e)
>> T = max(g(:));
>> g = g >= T;
>> figure, imshow(g) % Fig. 11.4(f)
```

The shades darker than the gray background in Fig. 11.4(b) correspond to negative values. There are two main segments oriented in the +45° direction, one at the top, left and one at the bottom, right [Figs. 11.4(c) and (d) show zoomed sections of these two areas]. Note how much brighter the straight line segment in Fig. 11.4(d) is than the segment in Fig. 11.4(c). The reason is that the component in the bottom, right of Fig. 11.4(a) is one pixel thick, while the one at the top, left is not. The mask response is stronger for the one-pixel-thick component.

Figure 11.4(e) shows the absolute value of Fig. 11.4(b). Because we are interested in the strongest response, we let `T` equal the maximum value in this image. Figure 11.4(f) shows in white the points whose values satisfied the condition `g >= T`, where `g` is the image in Fig. 11.4(e). The isolated points in this figure are points that also had similarly strong responses to the mask. In the original



a b
c d
e f

FIGURE 11.4
(a) Image of a wire-bond template. (b) Result of processing with the +45° detector in Fig. 11.3. (c) Zoomed view of the top, left region of (b). (d) Zoomed view of the bottom, right section of (b). (e) Absolute value of (b). (f) All points (in white) whose values satisfied the condition `g >= T`, where `g` is the image in (e). (The points in (f) were enlarged to make them easier to see.)

image, these points and their immediate neighbors are oriented in such a way that the mask produced a maximum response at those isolated locations. These isolated points can be detected using the mask in Fig. 11.1 and then deleted, or they could be deleted using morphological operators, as discussed in the last chapter. ■

11.1.3 Edge Detection Using Function edge

Although point and line detection certainly are important in any discussion on image segmentation, edge detection is by far the most common approach for detecting meaningful discontinuities in intensity values. Such discontinuities are detected by using first- and second-order derivatives. The first-order derivative of choice in image processing is the gradient, defined in Section 7.6.1. We repeat the pertinent equations here for convenience. The *gradient* of a 2-D function, $f(x, y)$, is defined as the *vector*

$$\nabla \mathbf{f} = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}$$

The magnitude of this vector is

$$\begin{aligned} \nabla f &= \text{mag}(\nabla \mathbf{f}) = \left[g_x^2 + g_y^2 \right]^{\frac{1}{2}} \\ &= \left[(\partial f/\partial x)^2 + (\partial f/\partial y)^2 \right]^{\frac{1}{2}} \end{aligned}$$

To simplify computation, this quantity is approximated sometimes by omitting the square-root operation,

$$\nabla f \approx g_x^2 + g_y^2$$

or by using absolute values,

$$\nabla f \approx |g_x| + |g_y|$$

These approximations still behave as derivatives; that is, they are zero in areas of constant intensity and their values are related to the degree of intensity change in areas of variable intensity. It is common practice to refer to the magnitude of the gradient or its approximations simply as "the gradient."

A fundamental property of the gradient vector is that it points in the direction of the maximum rate of change of f at coordinates (x, y). The angle at which this maximum rate of change occurs is

$$\alpha(x, y) = \tan^{-1}\left[\frac{g_y}{g_x} \right]$$

See the margin note in Section 7.6.1 regarding computation of the arctangent.

Methods for estimating g_x and g_y using function `edge` are discussed later in this section.

Second-order derivatives in image processing generally are computed using the Laplacian introduced in Section 3.5.1. Recall that the Laplacian of a 2-D function $f(x, y)$ is formed from second-order derivatives:

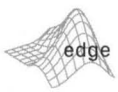
$$\nabla^2 f(x, y) = \frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2}$$

The Laplacian seldom is used directly for edge detection because, as a second-order derivative, it is unacceptably sensitive to noise, its magnitude produces double edges, and it is unable to detect edge direction. However, as discussed later in this section, the Laplacian can be a powerful complement when used in combination with other edge-detection techniques. For example, although its double edges make it unsuitable for edge detection, this property can be used for edge *location* by looking for zero crossings between double edges.

With the preceding discussion as background, the basic idea behind edge detection is to find places in an image where the intensity changes rapidly, using one of two general criteria:

1. Find places where the first derivative of the intensity is greater in magnitude than a specified threshold.
2. Find places where the second derivative of the intensity has a zero crossing.

Function `edge` in the Image Processing Toolbox provides several edge estimators based on the criteria just discussed. For some of these estimators, it is possible to specify whether the edge detector is sensitive to horizontal or vertical edges or to both. The general syntax for this function is




```
[g, t] = edge(f, 'method', parameters)
```

where `f` is the input image, `method` is one of the approaches listed in Table 11.1, and `parameters` are additional parameters explained in the following discussion. In the output, `g` is a logical array with 1s at the locations where edge points were detected in `f` and 0s elsewhere. Parameter `t` is optional; it gives the threshold used by `edge` to determine which gradient values are strong enough to be called edge points.

Sobel Edge Detector

First-order derivatives are approximated digitally by differences. The *Sobel edge detector* computes the gradient by using the following discrete differences between rows and columns of a 3×3 neighborhood [see Fig. Fig. 11.5(a)], where the center pixel in each row or column is weighted by 2 to provide smoothing (Gonzalez and Woods [2008]):

Edge Detector	Description
Sobel	Finds edges using the Sobel approximation to the derivatives in Fig. 11.5(b)
Prewitt	Finds edges using the Prewitt approximation to the derivatives in Fig. 11.5(c).
Roberts	Finds edges using the Roberts approximation to the derivatives in Fig. 11.5(d).
Laplacian of a Gaussian (LoG)	Finds edges by looking for zero crossings after filtering $f(x, y)$ with a Laplacian of a Gaussian filter.
Zero crossings	Finds edges by looking for zero crossings after filtering $f(x, y)$ with a specified filter.
Canny	Finds edges by looking for local maxima of the gradient of $f(x, y)$. The gradient is calculated using the derivative of a Gaussian filter. The method uses two thresholds to detect strong and weak edges, and includes the weak edges in the output only if they are connected to strong edges. Therefore, this method is more likely to detect true weak edges.

TABLE 11.1 Edge detectors available in function `edge`.

$$\begin{aligned}\nabla f &= \left[g_x^2 + g_y^2\right]^{\frac{1}{2}} \\ &= \left\{\left[(z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)\right]^2 \right. \\ &\quad \left. + \left[(z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)\right]^2\right\}^{\frac{1}{2}}\end{aligned}$$

where the z's are intensities. Then, we say that a pixel at location (x, y) is an edge pixel if $\nabla f \geq T$ at that location, where T is a specified threshold.

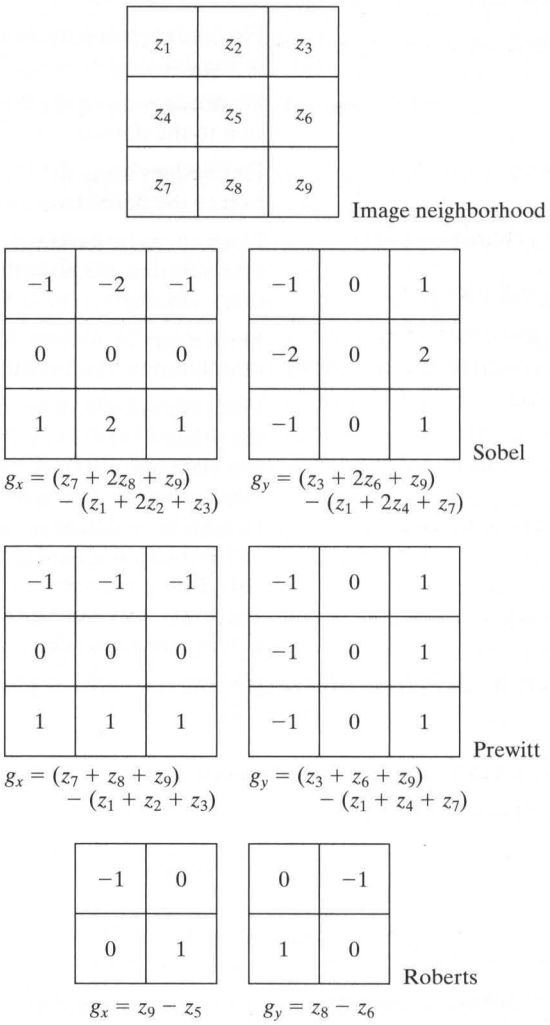
From the discussion in Section 3.5.1, we know that Sobel edge detection can be implemented by filtering an image, `f`, (using `imfilter`) with the left mask in Fig. 11.5(b), filtering `f` again with the other mask, squaring the pixels values of each filtered image, adding the two results, and computing their square root. Similar comments apply to the second and third entries in Table 11.1. Function `edge` simply packages the preceding operations into one function call and adds other features, such as accepting a threshold value or determining a threshold automatically. In addition, `edge` contains edge detection techniques that are not implementable directly with `imfilter`.

The general calling syntax for the Sobel detector is

```
[g, t] = edge(f, 'sobel', T, dir)
```

a
b
c
d

FIGURE 11.5
Edge detector masks and the first-order derivatives they implement.



where `f` is the input image, `T` is a specified threshold, and `dir` specifies the preferred direction of the edges detected: `'horizontal'`, `'vertical'`, or `'both'` (the default). As noted earlier, `g` is a logical image containing 1s at locations where edges were detected and 0s elsewhere. Parameter `t` in the output is optional. It is the threshold value used by `edge`. If `T` is specified, then `t = T`. If `T` is not specified (or is empty, `[ ]`), `edge` sets `t` equal to a threshold it determines automatically and then uses for edge detection. One of the principal reasons for including `t` in the output argument is to obtain an initial threshold value that can be modified and passed to the function in subsequent calls. Function `edge` uses the Sobel detector as a default if the syntax `g = edge(f)`, or `[g, t] = edge(f)`, is used.

Prewitt Edge Detector

The *Prewitt edge detector* uses the masks in Fig. 11.5(c) to approximate digitally the first derivatives g_x and g_y. Its general calling syntax is

```
[g, t] = edge(f, 'prewitt', T, dir)
```

The parameters of this function are identical to the Sobel parameters. The Prewitt detector is slightly simpler to implement computationally than the Sobel detector, but it tends to produce somewhat noisier results.

Roberts Edge Detector

The Roberts edge detector uses the masks in Fig. 11.5(d) to approximate digitally the first derivatives as differences between adjacent pixels. Its general calling syntax is

```
[g, t] = edge(f, 'roberts', T, dir)
```

The parameters of this function are identical to the Sobel parameters. The Roberts detector is one of the oldest edge detectors in digital image processing and, as Fig. 11.5(d) shows, it also is the simplest. This detector is used considerably less than the others in Fig. 11.5 due in part to its limited functionality (e.g., it is not symmetric and cannot be generalized to detect edges that are multiples of 45°). However, it still is used frequently in hardware implementations where simplicity and speed are dominant factors.

Laplacian of a Gaussian (LoG) Detector

Consider the Gaussian function

$$G(x, y) = e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

where σ is the standard deviation. This is a smoothing function which, if convolved with an image, will blur it. The degree of blurring is determined by the value of σ. The Laplacian of this function (see Gonzalez and Woods [2008]) is

$$\begin{aligned}\nabla^2 G(x, y) &= \frac{\partial^2 G(x, x)}{\partial x^2} + \frac{\partial^2 G(x, x)}{\partial y^2} \\ &= \left[\frac{x^2 + y^2 - 2\sigma^2}{\sigma^4}\right] e^{-\frac{x^2 + y^2}{2\sigma^2}}\end{aligned}$$

For obvious reasons, this function is called the *Laplacian of a Gaussian* (LoG). Because the second derivative is a linear operation, convolving (filtering) an image with $\nabla^2 G(x, y)$ is the same as convolving the image with the smoothing function first and then computing the Laplacian of the result. This is the key concept underlying the LoG detector. We convolve the image with $\nabla^2 G(x, y)$ knowing that it has two effects: It smooths the image (thus reducing noise),

and it computes the Laplacian, which yields a double-edge image. Locating edges then consists of finding the zero crossings between the double edges.

The general calling syntax for the LoG detector is

```
[g, t] = edge(f, 'log', T, sigma)
```

where `sigma` is the standard deviation and the other parameters are as explained previously. The default value for `sigma` is 2. As before, function `edge` ignores any edges that are not stronger than `T`. If `T` is not provided, or it is empty, `[ ]`, `edge` chooses the value automatically. Setting `T` to 0 produces edges that are closed contours, a familiar characteristic of the LoG method.

Zero-Crossings Detector

This detector is based on the same concept as the LoG method, but the convolution is carried out using a specified filter function, `H`. The calling syntax is

```
[g, t] = edge(f, 'zerocross', T, H)
```

The other parameters are as explained for the LoG detector.

Canny Edge Detector

The Canny detector (Canny [1986]) is the most powerful edge detector in function `edge`. The method can be summarized as follows:

1. The image is smoothed using a Gaussian filter with a specified standard deviation, σ, to reduce noise.
2. The local gradient, $[g_x^2 + g_y^2]^{\frac{1}{2}}$ and edge direction, $\tan^{-1}(g_x/g_y)$, are computed at each point. Any of the first three techniques in Table 11.1 can be used to compute the derivatives. An edge point is defined to be a point whose strength is locally maximum in the direction of the gradient.
3. The edge points determined in (2) give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output, a process known as *nonmaximal suppression*. The ridge pixels are then thresholded by so-called *hysteresis thresholding*, which is based on using two thresholds, T_1 and T_2, with $T_1 < T_2$. Ridge pixels with values greater than T_2 are said to be "strong" edge pixels. Ridge pixels with values between T_1 and T_2 are said to be "weak" edge pixels.
4. Finally, the algorithm performs edge linking by incorporating the weak pixels that are 8-connected to the strong pixels.

The syntax for the Canny edge detector is

```
[g, t] = edge(f, 'canny', T, sigma)
```

where `T` is a vector, `T = [T1, T2]`, containing the two thresholds explained in step 3 of the preceding procedure, and `sigma` is the standard deviation of the

smoothing filter. If `t` is included in the output argument, it is a two-element vector containing the two threshold values used by the algorithm. The rest of the syntax is as explained for the other methods, including the automatic computation of thresholds if `T` is not supplied. The default value for `sigma` is 1.

EXAMPLE 11.3: Using the Sobel edge detector.

■ We can extract and display the vertical edges in the image, `f`, of Fig. 11.6(a) using the commands

```
>> [gv, t] = edge(f, 'sobel', 'vertical');
>> imshow(gv)
>> t
t =
    0.0516
```

As Fig. 11.6(b) shows, the predominant edges in the result are vertical (the inclined edges have vertical and horizontal components, so they are detected as well). We can clean up the weaker edges somewhat by specifying a higher threshold value. For example, Fig. 11.6(c) was generated using the command

```
>> gv = edge(f, 'sobel', 0.15, 'vertical');
```

Using the same value of `T` in the command

```
>> gboth = edge(f, 'sobel', 0.15);
```

resulted in Fig. 11.6(d), which shows predominantly vertical and horizontal edges.

Function `edge` does not compute Sobel edges at $\pm 45°$. To compute such edges we need to specify the mask and use `imfilter`. For example, Fig. 11.6(e) was generated using the commands

```
>> wneg45 = [-2 -1 0; -1 0 1; 0 1 2]
weg45 =
    -2    -1     0
    -1     0     1
     0     1     2

>> gneg45 = imfilter(tofloat(f), wneg45, 'replicate');
>> T = 0.3*max(abs(gneg45(:)));
>> gneg45 = gneg45 >= T;
>> figure, imshow(gneg45);
```

The value of `T` was chosen experimentally to show results comparable with Figs. 11.6(c) and 11.6(d).

The strongest edge in Fig. 11.6(e) is the edge oriented at $-45°$. Similarly, using the mask `wpos45 = [0 1 2; -1 0 1; -2 -1 0]` with the same sequence of commands resulted in the strong edges oriented at $+45°$ in Fig. 11.6(f).

Using the `'prewitt'` and `'roberts'` options in function `edge` follows the same general procedure just illustrated for the Sobel edge detector. ■

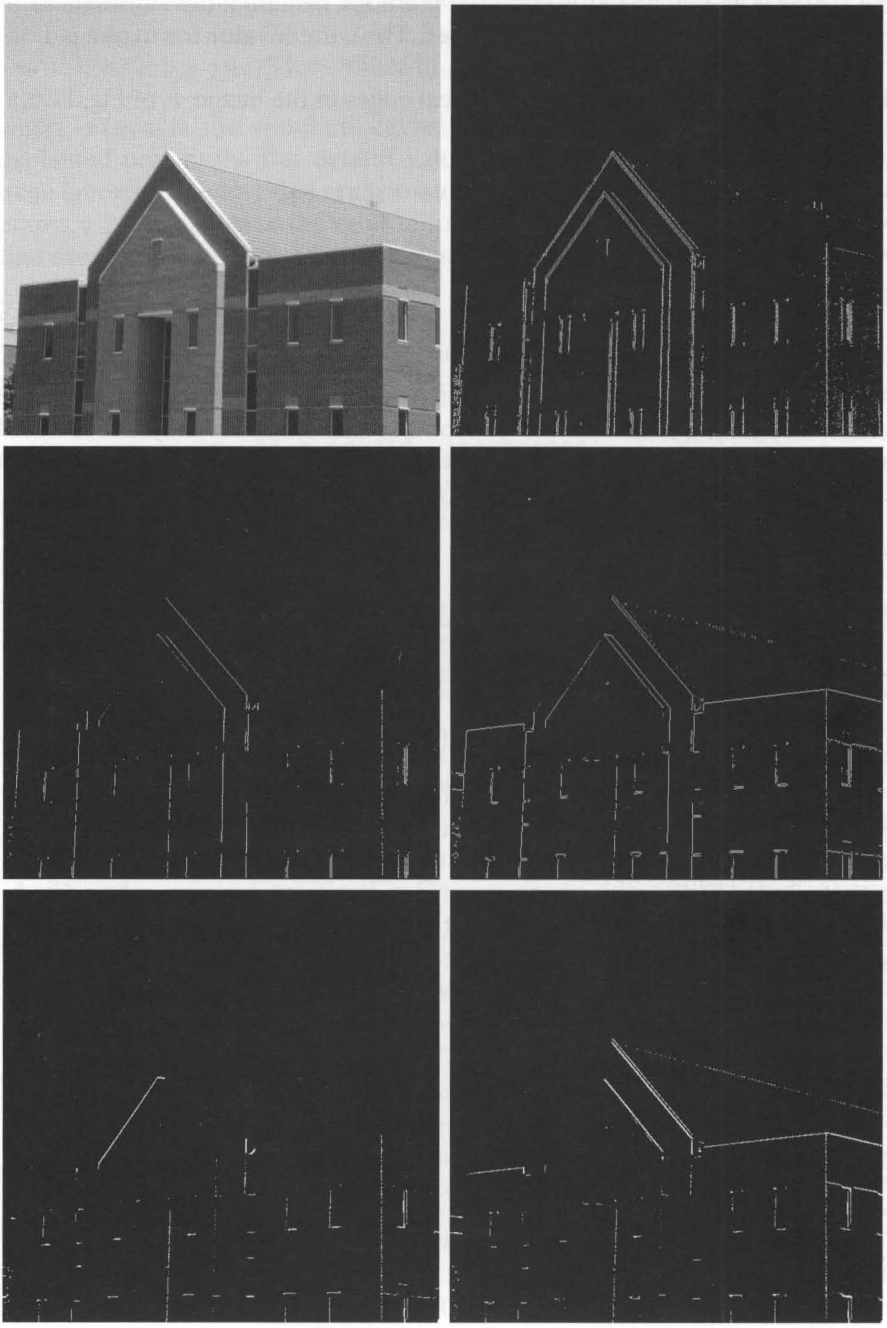


FIGURE 11.6
(a) Original image. (b) Result of function `edge` using a vertical Sobel mask with the threshold determined automatically. (c) Result using a specified threshold. (d) Result of determining both vertical and horizontal edges with a specified threshold. (e) Result of computing edges at $-45°$ with `imfilter` using a specified mask and a specified threshold. (f) Result of computing edges at $+45°$ with `imfilter` using a specified mask and a specified threshold.

EXAMPLE 11.4: Comparison of the Sobel, LoG, and Canny edge detectors.

■ In this example we compare the relative performance of the Sobel, LoG, and Canny edge detectors. The objective is to produce a clean *edge map* by extracting the principal edge features of the building image, `f`, in Fig. 11.6(a), while reducing "irrelevant" detail, such as the fine texture in the brick walls and tile roof. The principal features of interest in this discussion are the edges forming the building corners, the windows, the light-brick structure framing the entrance, the entrance itself, the roof line, and the concrete band surrounding the building about two-thirds of the distance above ground level.

The left column in Fig. 11.7 shows the edge images obtained using the default syntax for the `'sobel'`, `'log'`, and `'canny'` options:

```
>> f = tofloat(f);
>> [gSobel_default, ts] = edge(f, 'sobel');  % Fig. 11.7(a)
>> [gLoG_default, tlog] = edge(f, 'log');    % Fig. 11.7(c)
>> [gCanny_default, tc] = edge(f, 'canny'); % Fig. 11.7(e)
```

The values of the thresholds in the output argument resulting from the preceding computations were `ts = 0.074`, `tlog = 0.0025`, and `tc = [0.019, 0.047]`. The defaults values of `sigma` for the `'log'` and `'canny'` options were 2.0 and 1.0, respectively. With the exception of the Sobel image, the default results were far from the objective of producing clean edge maps.

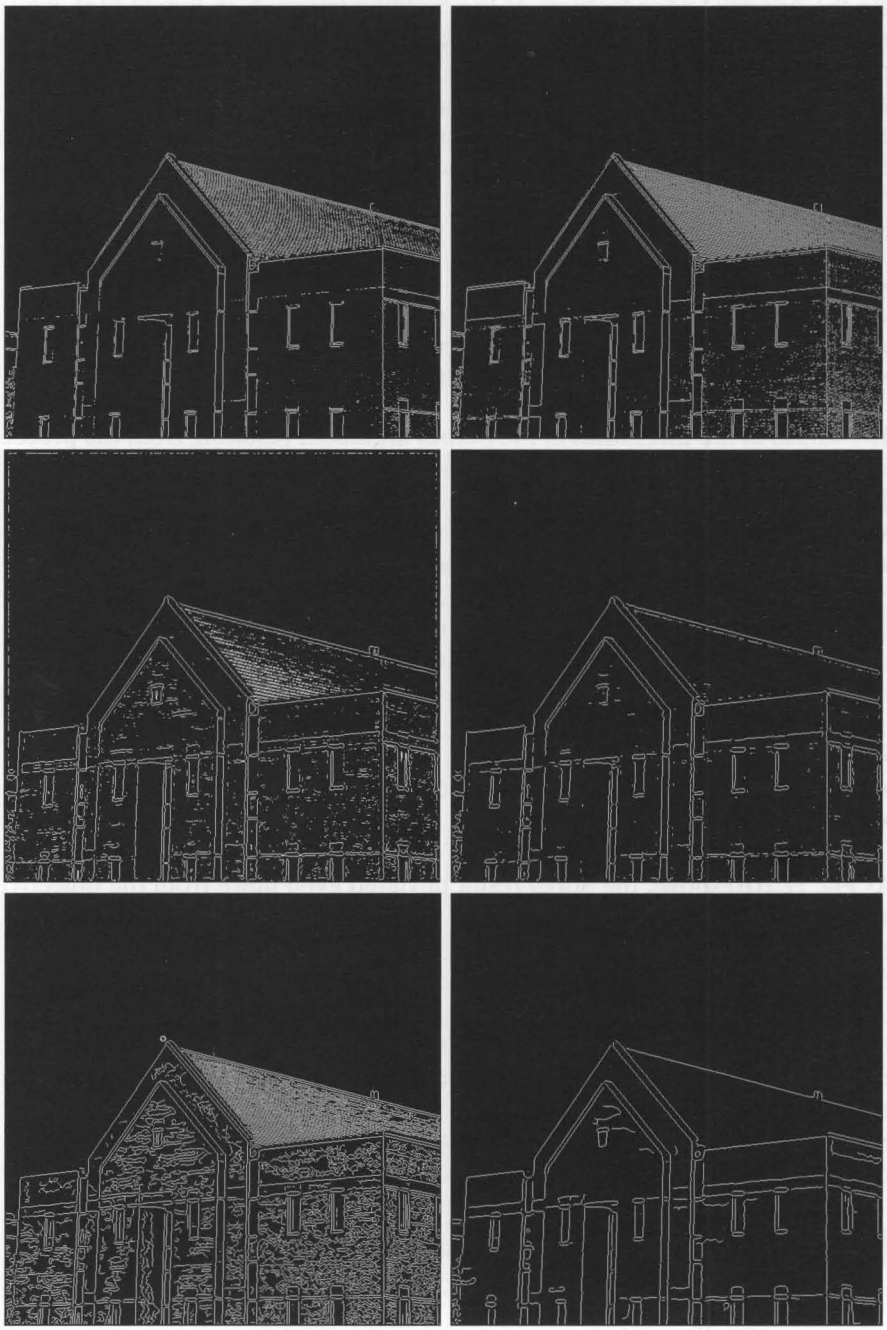
Starting with the default values, the parameters in each option were varied interactively with the objective of bringing out the principal features mentioned earlier, while reducing irrelevant detail as much as possible. The results in the right column of Fig. 11.7 were obtained with the following commands:

```
>> gSobel_best = edge(f, 'sobel', 0.05); % Fig. 11.7(b)
>> gLoG_best = edge(f, 'log', 0.003, 2.25); % Fig. 11.7(d)
>> gCanny_best = edge(f, 'canny', [0.04 0.10], 1.5); % Fig. 11.7(f)
```

As Fig. 11.7(b) shows, the Sobel result deviated even more from the objective when we tried to detect both edges of the concrete band and the left edge of the entrance. The LoG result in Fig. 11.7(d) is somewhat better than the Sobel result and much better than the LoG default, but it still could not detect the left edge of the main entrance, nor both edges of the concrete band. The Canny result [Fig. 11.7(f)] is superior by far to the other two results. Note in particular how the left edge of the entrance was clearly detected, as were both edges of the concrete band, and other details such as the roof ventilation grill above the main entrance. In addition to detecting the desired features, the Canny detector also produced the cleanest edge map. ■

11.2 Line Detection Using the Hough Transform

Ideally, the methods discussed in the previous section should yield pixels lying only on edges. In practice, the resulting pixels seldom characterize an edge completely because of noise, breaks in the edge from nonuniform illumination, and other effects that introduce spurious intensity discontinuities. Thus, edge-



a b
c d
e f

FIGURE 11.7
Left column: Default results for the Sobel, LoG, and Canny edge detectors. Right column: Results obtained interactively to bring out the principal features in the original image of Fig. 11.6(a), while reducing irrelevant detail. The Canny edge detector produced the best result.

detection algorithms typically are followed by linking procedures to assemble edge pixels into meaningful edges. One approach for linking line segments in an image is the *Hough transform* (Hough [1962]).

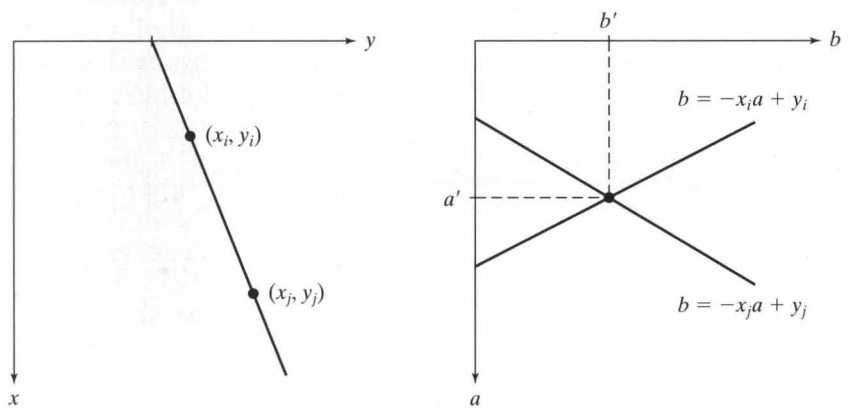
11.2.1 Background

Given n points in an image (typically a binary image), suppose that we want to find subsets of these points that lie on straight lines. One possible solution is to first find all lines determined by every pair of points and then find all subsets of points that are close to particular lines. The problem with this procedure is that it involves finding $n(n-1)/2 \sim n^2$ lines and then performing $n(n(n-1))/2 \sim n^3$ comparisons of every point to all lines. This approach is computationally prohibitive in all but the most trivial applications.

With the Hough transform, on the other hand, we consider a point (x_i, y_i) and all the lines that pass through it. Infinitely many lines pass through (x_i, y_i), all of which satisfy the *slope-intercept* line equation $y_i = ax_i + b$ for some values of a and b. Writing this equation as $b = -ax_i + y_i$ and considering the ab-plane (also called *parameter space*) yields the equation of a *single* line for a fixed pair (x_i, y_i). Furthermore, a second point (x_j, y_j) also has a line in parameter space associated with it, and this line intersects the line associated with (x_i, y_i) at (a', b') where a' is the slope and b' the intercept of the line containing both (x_i, y_i) and (x_j, y_j) in the xy-plane. In fact, all points contained on this line have lines in parameter space that intersect at (a', b'). Figure 11.8 illustrates these concepts.

In principle, the parameter-space lines corresponding to all image points (x_k, y_k) in the xy-plane could be plotted, and the principal lines in that plane could be found by identifying points in parameter space where large number of parameter-space lines intersect. However, a practical difficulty with this approach is that a (the line slope) approaches infinity as the line approaches the vertical direction. One way around this difficulty is to use the *normal representation* of a line:

$$x\cos\theta + y\sin\theta = \rho$$



a b

FIGURE 11.8
(a) xy-plane. (b) Parameter space.

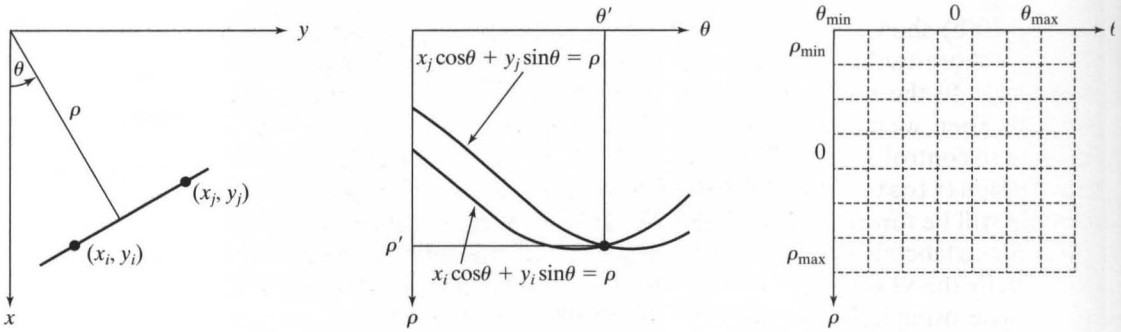
We follow convention in the way we show the angle in Fig. 11.9(a). However, the toolbox references θ with respect to the positive horizontal axis (with positive angles measured in the clockwise direction) and limits the range to $[-90°, 90°]$. For example, an angle of $-16°$ in our figure would correspond to an angle of $116°$ in the toolbox. The toolbox brings this angle into the allowed range by performing the operation $106° - 180° = -74°$.

Figure 11.9(a) illustrates the geometric interpretation of the parameters ρ and θ. A horizontal line has $\theta = 0°$, with ρ being equal to the positive x-intercept. Similarly, a vertical line has $\theta = 90°$, with ρ being equal to the positive y-intercept, or $\theta = -90°$, with ρ being equal to the negative y intercept. Each sinusoidal curve in Fig. 11.9(b) represents the family of lines that pass through a particular point (x_i, y_i). The intersection point (ρ', θ') corresponds to the line that passes through both (x_i, y_i) and (x_j, y_j).

The computational attractiveness of the Hough transform arises from subdividing the $\rho\theta$ parameter space into so-called *accumulator cells*, as illustrated in Fig. 11.9(c), where $[\rho_{\min}, \rho_{\max}]$ and $[\theta_{\min}, \theta_{\max}]$ are the expected ranges of the parameter values. Usually, the maximum range of values is $-D \le \rho \le D$ and $-90° \le \theta \le 90°$, where D is the farthest distance between opposite corners in the image. The cell at coordinates (i, j) with accumulator value $A(i, j)$ corresponds to the square associated with parameter space coordinates (ρ_i, θ_j). Initially, these cells are set to zero. Then, for every nonbackground point (x_k, y_k) in the image plane (i.e., the xy-plane), we let θ equal each of the allowed subdivision values on the θ-axis and solve for the corresponding ρ using the equation $\rho = x_k \cos\theta + y_k \sin\theta$. The resulting ρ-values are then rounded off to the nearest allowed cell value along the ρ-axis. The corresponding accumulator cell is then incremented. At the end of this procedure, a value of Q in cell $A(i, j)$ means that Q points in the xy-plane lie on the line $x\cos\theta_j + y\sin\theta_j = \rho_i$. The number of subdivisions in the $\rho\theta$-plane determines the accuracy of the colinearity of these points. The accumulator array is referred to in the toolbox as the *Hough transform matrix*, or simply as the *Hough transform*.

11.2.2 Toolbox Hough Functions

The Image Processing Toolbox provides three functions related to the Hough transform. Function `hough` implements the concepts in the previous section, function `houghpeaks` finds the peaks (high-count accumulator cells) in the



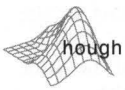
a b c

FIGURE 11.9 (a) Parameterization of lines in the xy-plane. (b) Sinusoidal curves in the $\rho\theta$-plane; the point of intersection, (ρ', θ'), corresponds to the parameters of the line joining (x_i, y_i) and (x_j, y_j). (c) Division of the $\rho\theta$-plane into accumulator cells.

Hough transform, and function `houghlines` extracts line segments in the original image based on the results from the other two functions.

Function hough

Function `hough` has either the default syntax




```
[H, theta, rho] = hough(f)
```

or the complete syntax form

```
[H, theta, rho] = hough(f, 'ThetaRes', val1, 'RhoRes', val2)
```

where `H` is the Hough transform matrix, and `theta` (in degrees) and `rho` are the vectors of θ and ρ values over which the Hough transform matrix was generated. Input `f` is a *binary* image; `val1` is a scalar between 0 and 90 that specifies the Hough transform bins along the θ-axis (the default is 1); and `val2` is a real scalar in the range `0 < val2 < hypot(size(I, 1), size(I, 2))` that specifies the spacing of the Hough transform bins along the ρ-axis (the default is 1).

EXAMPLE 11.5: Illustration of the Hough transform.

■ In this example we illustrate the mechanics of function `hough` using a simple synthetic image:

```
>> f = zeros(101, 101);
>> f(1, 1)     = 1;  f(101, 1) = 1; f(1, 101) = 1;
>> f(101, 101) = 1;  f(51, 51) = 1;
```

Figure 11.10(a) shows our test image. Next, we compute and display the Hough transform using the defaults:

```
>> H = hough(f);
>> imshow(H, [ ])
```

Figure 11.10(b) shows the result, displayed with `imshow` in the familiar way. Often, it is more useful to visualize Hough transforms in a larger plot, with labeled axes. In the next code fragment we call `hough` using all the output arguments. Then, we pass vectors `theta` and `rho` as additional input arguments to `imshow` to control the horizontal and vertical axis labeling. We also pass the `'InitialMagnification'` option to `imshow` with value `'fit'` so that the entire image will be forced to fit in the figure window. The `axis` function is used to turn on axis labeling and to make the display fill the rectangular shape of the figure. Finally the `xlabel` and `ylabel` functions (see Section 3.3.1) are used to label the axes using LaTeX-style notation for the Greek letters:

```
>> [H, theta, rho] = hough(f);
>> imshow(H, [], 'XData', theta, 'YData', rho,'InitialMagnification', 'fit')
>> axis on, axis normal
>> xlabel('\theta'), ylabel('\rho')
```

a b
c

FIGURE 11.10
(a) Binary image with five dots (four of the dots are in the corners).
(b) Hough transform displayed using `imshow`.
(c) Alternative Hough transform display with axis labeling. [The dots in (a) were enlarged to make them easier to see.]

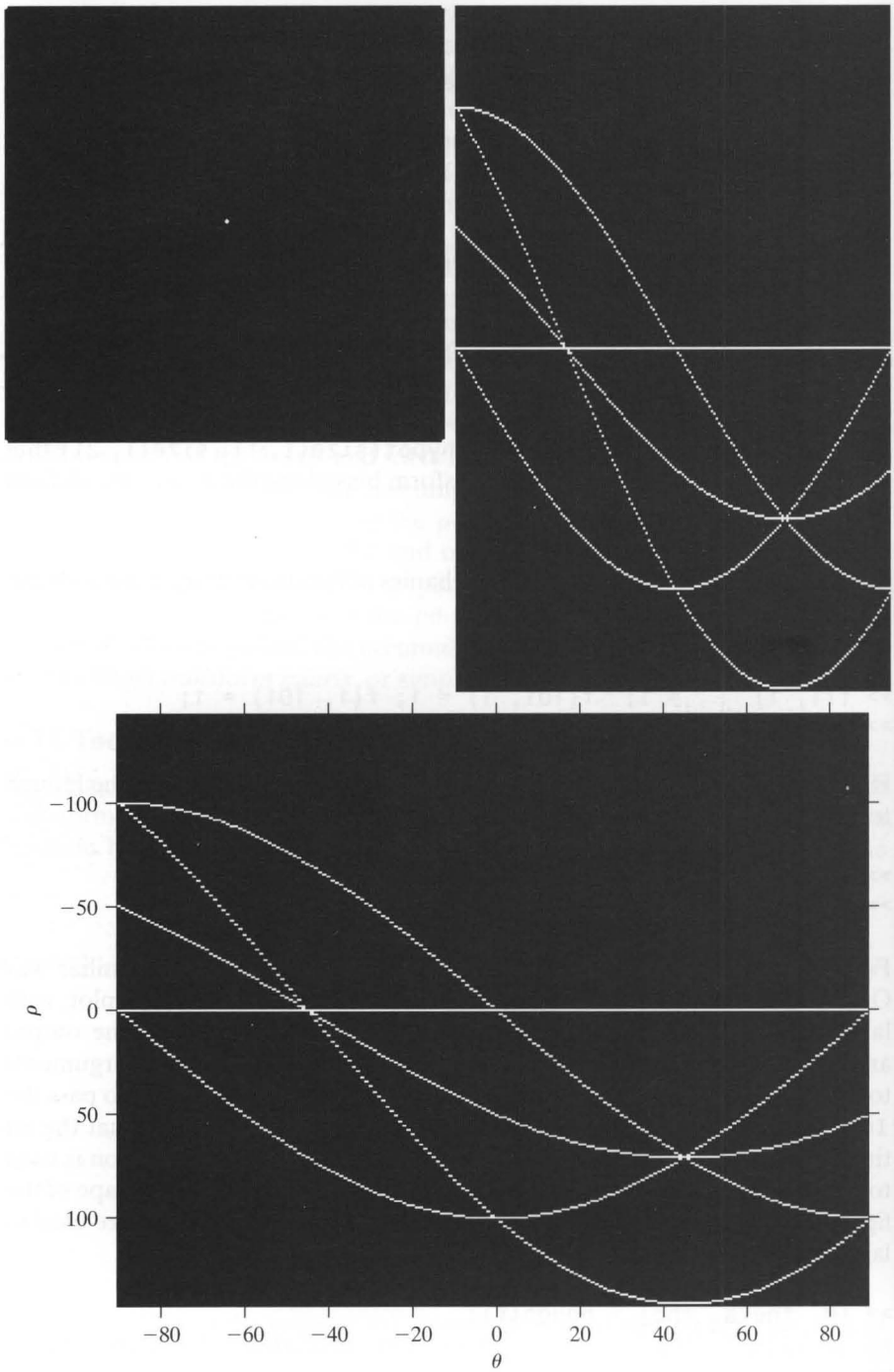


Figure 11.10(c) shows the labeled result. The intersections of three curves (the straight line is a considered a curve also) at $\pm 45°$ indicate that there are two sets of three collinear points in `f`. The intersections of two curves at $(\rho, \theta) = (0,-90), (-100,-90), (0,0)$, and $(100,0)$ indicate that there are four sets of collinear points that lie along vertical and horizontal lines. ■

Function houghpeaks

The first step in using the Hough transform for line detection and linking is to find accumulator cells with high counts (toolbox documentation refers to high cell values as *peaks*). Because of the quantization in parameter space of the Hough transform, and the fact that edges in typical images are not perfectly straight, Hough transform peaks tend to lie in more than one Hough transform cell. Function `houghpeaks` finds a specified number of peaks (`NumPeaks`) using either the default syntax:

```
peaks = houghpeaks(H, NumPeaks)
```



or the complete syntax form

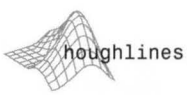
```
peaks = houghpeaks(..., 'Threshold', val1, 'NHoodSize', val2)
```

where "..." indicates the inputs from the default syntax and `peaks` is a $Q \times 2$ matrix holding the row and column coordinates of the peaks; Q can range from `0` to `NumPeaks`. `H` is the Hough transform matrix. Parameter `val1` is a nonnegative scalar that specifies which values in `H` are considered peaks; `val1` can vary from `0` to `Inf`, the default being `0.5*max(H(:))`. Parameter `val2` is a two-element vector of odd integers that specifies a neighborhood size around each peak. The elements in the neighborhood are set to zero after the peak is identified. The default is the two-element vector consisting of the smallest odd values greater than or equal to `size(H)/50`. The basic idea behind this procedure is to clean-up the peaks by setting to zero the Hough transform cells in the immediate neighborhood in which a peak was found. We illustrate function `houghpeaks` in Example 11.6.

Function houghlines

Once a set of candidate peaks has been identified in the Hough transform, it remains to be determined if there are meaningful line segments associated with those peaks, as well as where the lines start and end. Function `houghlines` performs this task using either its default syntax

```
lines = houghlines(f, theta, rho, peaks)
```



or the complete syntax form

```
lines = houghlines(..., 'FillGap', val1, 'MinLength', val2)
```

where `theta` and `rho` are outputs from function `hough`, and `peaks` is the output of function `houghpeaks`. Output `lines` is a structure array whose length equals the number of line segments found. Each element of the structure identifies one line, and has the following fields:

- `point1`, a two-element vector `[r1, c1]` specifying the row and column coordinates of one end point of the line segment.
- `point2`, a two-element vector `[r2, c2]` specifying the row and column coordinates of the other end point of the line segment.
- `theta`, the angle in degrees of the Hough transform bin associated with the line.
- `rho`, the ρ-axis position of the Hough transform bin associated with the line.

The other parameters are as follows: `val1` is a positive scalar that specifies the distance between two line segments associated with the same Hough transform bin. When the distance between the line segments is less than the value specified, function `houghlines` merges the line segments into a single segment (the default distance is 20 pixels). Parameter `val2` is a positive scalar that specifies whether merged lines should be kept or discarded. Lines shorter than the value specified in `val2` are discarded (the default is 40).

EXAMPLE 11.6: Using the Hough transform for line detection and linking.

■ In this example we use functions `hough`, `houghpeaks`, and `houghlines` to find a set of line segments in the binary image, `f`, in Fig. 11.7(f). First, we compute and display the Hough transform, using a finer angular spacing than the default (0.2 instead of 1.0):

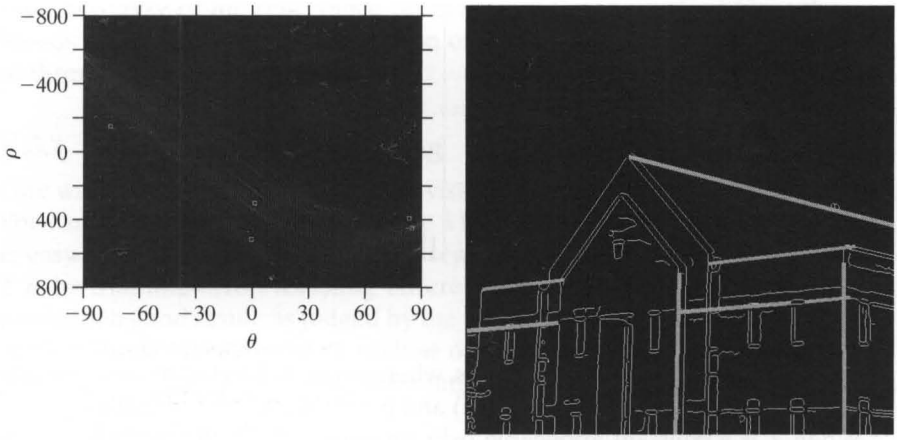
```
>> [H, theta, rho] = hough(f, 'ThetaResolution', 0.2);
>> imshow(H, [], 'XData', theta, 'YData', rho, 'InitialMagnification', 'fit')
>> axis on, axis normal
>> xlabel('\theta'), ylabel('\rho')
```

Next we use function `houghpeaks` to find, say, five significant Hough transform peaks:

```
>> peaks = houghpeaks(H, 5);
>> hold on
>> plot(theta(peaks(:, 2)), rho(peaks(:, 1)), ...
         'linestyle', 'none', 'marker', 's', 'color', 'w')
```

The preceding operations compute and display the Hough transform and superimpose the locations of five peaks found using the default settings of function `houghpeaks`. Figure 11.11(a) shows the results. For example, the leftmost small square identifies the accumulator cell associated with the roof, which is inclined at approximately $-74°$ in the toolbox angle reference [$-16°$ in Fig. 11.9(a)—see the margin note related to that figure for an explanation of the Hough angle convention used by the toolbox.]

Finally, we use function `houghlines` to find and link line segments, and then superimpose the line segments on the original binary image using functions `imshow`, `hold on`, and `plot`:



a b

FIGURE 11.11
(a) Hough transform with five peak locations selected. (b) Line segments (in bold) corresponding to the Hough transform peaks.

```
>> lines = houghlines(f, theta, rho, peaks);
>> figure, imshow(f), hold on
>> for k = 1:length(lines)
xy = [lines(k).point1 ; lines(k).point2];
plot(xy(:,1), xy(:,2), 'LineWidth', 4, 'Color', [.8 .8 .8]);
end
```

Figure 11.11(b) shows the resulting image with the detected segments superimposed as thick, gray lines. ■

11.3 Thresholding

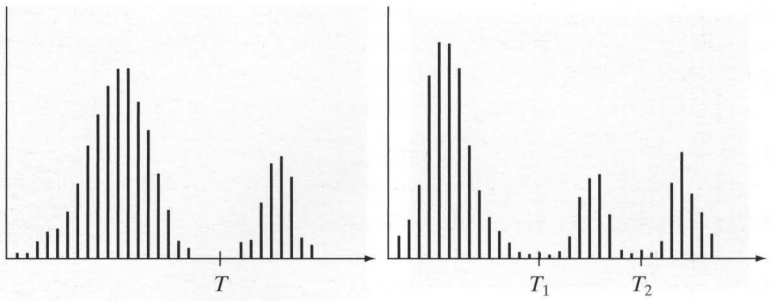
Because of its intuitive properties and simplicity of implementation, image thresholding enjoys a central position in applications of image segmentation. Simple thresholding was first introduced in Section 2.7, and we have used it in various discussions in the preceding chapters. In this section, we discuss ways of choosing the threshold value automatically, and we consider a method for varying the threshold based on local image properties.

11.3.1 Foundation

Suppose that the intensity histogram shown in Fig. 11.12(a) corresponds to an image, $f(x, y)$, composed of light objects on a dark background, in such a way that object and background pixels have intensity levels grouped into two dominant modes. One obvious way to extract the objects from the background is to select a threshold T that separates these modes. Then any image point (x, y) at which $f(x, y) > T$ is called an *object (*or *foreground) point*; otherwise, the point is called a *background point* (the reverse holds for dark objects on a light background). The thresholded (binary) image $g(x, y)$ is defined as

We use the terms *object point* and *foreground point* interchangeably.

$$g(x, y) = \begin{cases} a & \text{if } f(x, y) > T \\ b & \text{if } f(x, y) \leq T \end{cases}$$



a b

FIGURE 11.12 Intensity histograms that can be partitioned (a) by a single threshold, and (b) by dual thresholds. These are *unimodal* and *bimodal* histograms, respectively.

Pixels labeled a correspond to objects, whereas pixels labeled b correspond to the background. Usually, $a = 1$ (white) and $b = 0$ (black) by convention.

When T is a constant applicable over an entire image, the preceding equation is referred to as *global thresholding*. When the value of T changes over an image, we use the term *variable thresholding*. The term *local* or *regional thresholding* is used also to denote variable thresholding in which the value of T at any point (x, y) in an image depends on properties of a neighborhood of (x, y) (for example, the average intensity of the pixels in the neighborhood). If T depends on the spatial coordinates (x, y) themselves, then variable thresholding is often referred to as *dynamic* or *adaptive* thresholding. Use of these terms is not universal, and you are likely to see them used interchangeably in the literature on image processing.

Figure 11.12(b) shows a more difficult thresholding problem involving a histogram with three dominant modes corresponding, for example, to two types of light objects on a dark background. Here, *multiple* (*dual*) *thresholding* classifies a pixel at (x, y) as belonging to the background if $f(x, y) \leq T_1$, to one object class if $T_1 < f(x, y) \leq T_2$, and to the other object class if $f(x, y) > T_2$. That is, the segmented image is given by

$$g(x, y) = \begin{cases} a & \text{if } f(x, y) > T_2 \\ b & \text{if } T_1 < f(x, y) \leq T_2 \\ c & \text{if } f(x, y) \leq T_1 \end{cases}$$

where a, b, and c are three distinct intensity values. Segmentation problems requiring more than two thresholds are difficult (often impossible) to solve, and better results usually are obtained using other methods, such as variable thresholding, as discussed in Sections 11.3.6 and 11.3.7, or region growing, as discussed in Section 11.4.

Based on the preceding discussion, we conclude that the success of intensity thresholding is related directly to the width and depth of the valley(s) separating the histogram modes. In turn, the key factors affecting the properties of the valley(s) are: (1) the separation between peaks (the further apart the peaks are, the better the chances of separating the modes); (2) the noise content in the image (the modes broaden as noise increases); (3) the relative sizes of objects and background; (4) the uniformity of the illumination source; and (5)

the uniformity of the reflectance properties of the image (see Gonzalez and Woods [2008] for a detailed discussion on how these factors affect the success of thresholding methods.

11.3.2 Basic Global Thresholding

One way to choose a threshold is by visual inspection of the image histogram. For example, the histogram in Figure 11.12(a) has two distinct modes, and it is easy to choose a threshold T that separates them. Another way to choose T is by trial and error, selecting different thresholds until one is found that produces a good result, as judged by the observer. This is particularly effective in an interactive environment, such as one that allows the user to change the threshold using a *widget* (graphical control, such as a slider) and see the result immediately.

Generally in image processing, the preferred approach is to use an algorithm capable of choosing a threshold automatically based on image data. The following iterative procedure is one such approach:

1. Select an initial estimate for the global threshold, T.
2. Segment the image using T. This will produce two groups of pixels: G_1, consisting of all pixels with intensity values greater than T and G_2, consisting of pixels with values less than or equal to T.
3. Compute the average intensity values m_1 and m_2 for the pixels in regions G_1 and G_2, respectively.
4. Compute a new threshold value:

$$T = \frac{1}{2}(m_1 + m_2)$$

5. Repeat steps 2 through 4 until the difference in T in successive iterations is smaller than a predefined value, ΔT.
6. Segment the image using function `im2bw`:

```
g = im2bw(f, T/den)
```

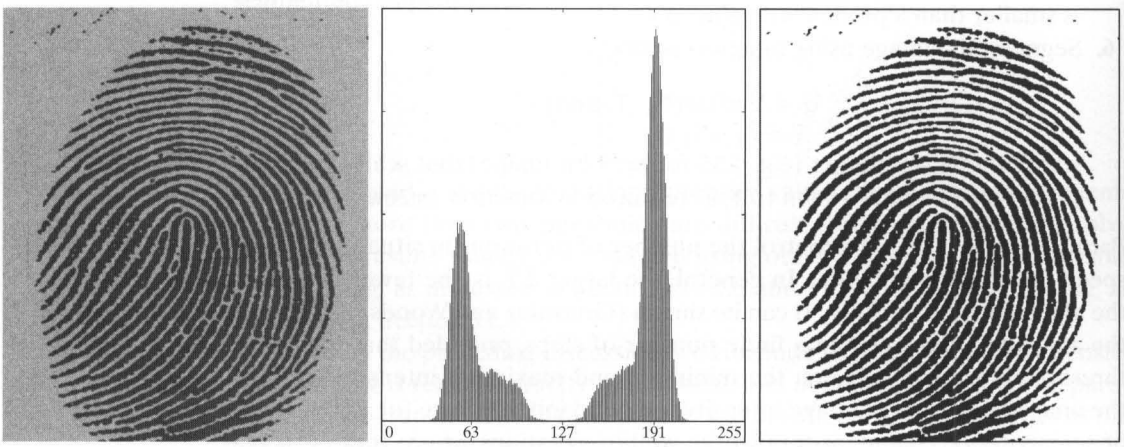
where `den` is an integer (e.g., `255` for an 8-bit image) that scales the maximum value of ratio `T/den` to `1`, as required by function `im2bw`.

Parameter ΔT is used to control the number of iterations in situations where speed is an important issue. In general, the larger ΔT is, the fewer iterations the algorithm will perform. It can be shown (Gonzalez and Woods [2008]) that the algorithm converges in a finite number of steps, provided that the initial threshold is chosen between the minimum and maximum intensity levels in the image (the average image intensity is a good initial choice for T). In terms of segmentation, the algorithm works well in situations where there is a reasonably clear valley between the modes of the histogram related to objects and background. We show how to implement this procedure in MATLAB in the following example.

EXAMPLE 11.7: Computing a global threshold.

■ The basic iterative method just developed can be implemented as follows, where `f` is the image in Fig. 11.13(a):

```
>> count = 0;
>> T = mean2(f);
>> done = false;
>> while ~done
     count = count + 1;
     g = f > T;
     Tnext = 0.5*(mean(f(g)) + mean(f(~g)));
     done = abs(T - Tnext) < 0.5;
     T = Tnext;
end
>> count
count =
     2
>> T
T =
    125.3860
>> g = im2bw(f, T/255);
>> imshow(f) % Fig. 11.13(a).
>> figure, imhist(f) % Fig. 11.13(b).
>> figure, imshow(g) % Fig. 11.13(c).
```



a b c

FIGURE 11.13 (a) Noisy fingerprint. (b) Histogram. (c) Segmented result using a global threshold (the border was added manually for clarity). (Original courtesy of the National Institute of Standards and Technology.)

The algorithm converged in only two iterations, and resulted in a threshold value near the midpoint of the gray scale. A clean segmentation was expected, because of the wide separation between modes in the histogram. ■

11.3.3 Optimum Global Thresholding Using Otsu's Method

Let the components of an image histogram be denoted by

$$p_q = \frac{n_q}{n} \qquad q = 0, 1, 2, \ldots, L-1$$

where n is the total number of pixels in the image, n_q is the number of pixels that have intensity level q, and L is the total number of possible intensity levels in the image (remember, intensity levels are integer values). Now, suppose that a threshold k is chosen such that C_1 is the set of pixels with levels $[0, 1, 2, \ldots, k]$ and C_2 is the set of pixels with levels $[k + 1, \ldots, L - 1]$. Otsu's method (Otsu [1979]) is optimum, in the sense that it chooses the threshold value k that maximizes the *between-class variance* $\sigma_B^2(k)$, defined as

$$\sigma_B^2(k) = P_1(k)\left[m_1(k) - m_G\right]^2 + P_2(k)\left[m_2(k) - m_G\right]^2$$

Here, $P_1(k)$ is the probability of set C_1 occurring:

$$P_1(k) = \sum_{i=0}^{k} p_i$$

For example, if we set $k = 0$, the probability of set C_1 having any pixels assigned to it is zero. Similarly, the probability of set C_2 occurring is

$$P_2(k) = \sum_{i=k+1}^{L-1} p_i = 1 - P_1(k)$$

The terms $m_1(k)$ and $m_2(k)$ are the mean intensities of the pixels in sets C_1 and C_2, respectively. The term m_G is the global mean (the mean intensity of the entire image):

$$m_G = \sum_{i=0}^{L-1} ip_i$$

Also, the mean intensity up to level k is given by

$$m(k) = \sum_{i=0}^{k} ip_i$$

By expanding the expression for $\sigma_B^2(k)$, and using the fact that $P_2(k) = 1 - P_1(k)$, we can write the between-class variance as

$$\sigma_B^2(k) = \frac{\left[m_G P_1(k) - m(k)\right]^2}{P_1(k)\left[1 - P_1(k)\right]}$$

This expression is slightly more efficient computationally because only two parameters, m and P_1 have to be computed for all values of k (m_G is computed only once).

The idea of maximizing the between-class variance is that the larger this variance is, the more likely it is that the threshold will segment the image properly. Note that this optimality measure is based entirely on parameters that can be obtained *directly* from the image histogram. In addition, because k is an integer in the range $[0, L-1]$, finding the maximum of $\sigma_B^2(k)$ is straightforward: We simply step through all L possible values of k and compute the variance at each step. We then select the k that gave the largest value of $\sigma_B^2(k)$. That value of k is the optimum threshold. If the maximum is not unique, the threshold used is the average of all the optimum k's found.

The ratio of the between-class variance to the total image intensity variance,

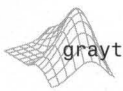
$$\eta(k) = \frac{\sigma_B^2(k)}{\sigma_G^2}$$

is a measure of the separability of image intensities into two classes (e.g., objects and background), which can be shown to be in the range

$$0 \leq \eta(k^*) \leq 1$$

where k^* is the optimum threshold. The measure achieves its minimum value for constant images (whose pixels are completely inseparable into two classes) and its maximum value for binary images (whose pixels are totally separable).

Toolbox function `graythresh` computes Otsu's threshold. It's syntax is




```
[T, SM] = graythresh(f)
```

where `f` is the input image, `T` is the resulting threshold, normalized to the range [0, 1], and `SM` is the separability measure. The image is segmented using function `im2bw`, as explained in the previous section.

EXAMPLE 11.8: Comparison of image segmentation using Otsu's method and the basic global thresholding technique from Section 11.3.2.

■ We begin by comparing Otsu's method with the global thresholding technique from the last section, using image `f` in Fig. 11.13(a):

```
>> [T, SM] = graythresh(f)
T =
    0.4902
SM =
    0.9437
```

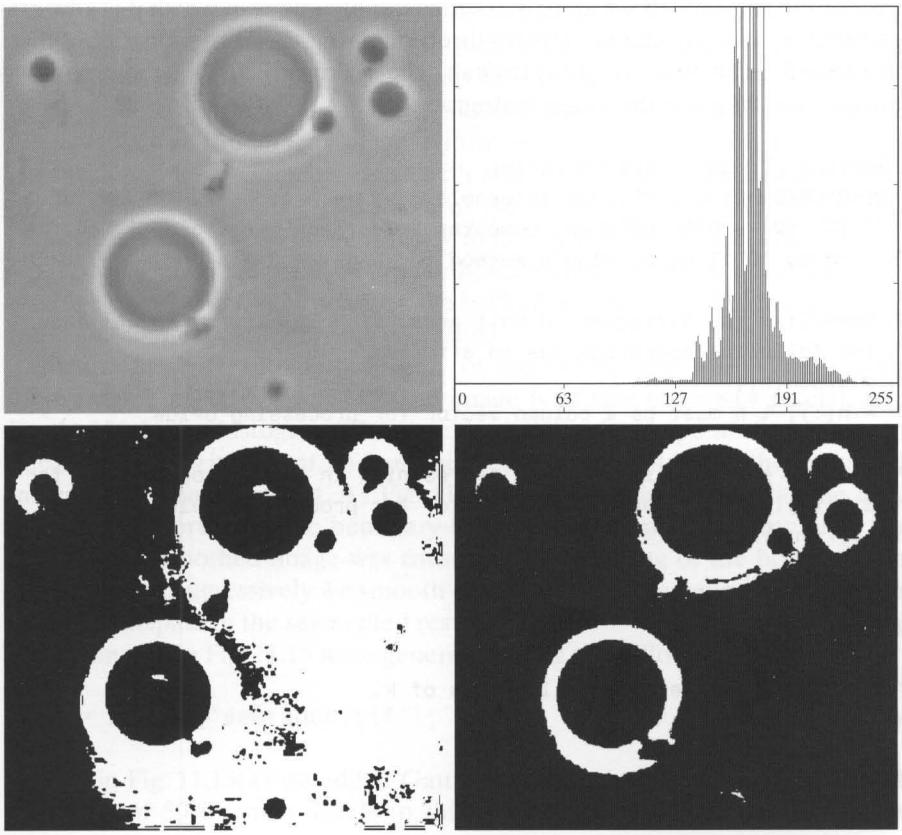
```
>> T*255
ans =
   125
```

This threshold has nearly the same value as the threshold obtained using the basic global thresholding algorithm from the last section, so we would expect the same segmentation result. Note the high value of `SM`, indicating a high degree of separability of the intensities into two classes.

Figure 11.14(a) (an image of polymersome cells, which we call `f2`) presents a more difficult segmentation task. The objective is to segment the borders of the cells (the brightest regions in the image) from the background. The image histogram [Fig. 11.14(b)] is far from bimodal, so we would expect the simple algorithm from the last section to have difficulty in achieving a suitable segmentation. The image in Fig. 11.14(c) was obtained using the same procedure that we used to obtain Fig. 11.13(c). The algorithm converged in one iteration and yielded a threshold, `T`, equal to 169.4. Using this threshold,

Polymersomes are cells artificially engineered using polymers. Polymersomes are invisible to the human immune system and can be used, for example, to deliver medication to targeted regions of the body.

```
>> g = im2bw(f2, T/255);
>> imshow(g)
```



a b
c d

FIGURE 11.14
(a) Original image.
(b) Histogram (high values were clipped to highlight details in the lower values).
(c) Segmentation result using the basic global algorithm from Section 11.3.2.
(d) Result obtained using Otsu's method.
(Original image courtesy of Professor Daniel A. Hammer, the University of Pennsylvania.)

resulted in Fig. 11.14(c). As you can see, the segmentation was unsuccessful.

We now segment the image using Otsu's method:

```
>> [T, SM] = graythresh(f2);
>> SM
SM =
    0.4662
>> T*255
ans =
   181
>> g = im2bw(f2, T);
>> figure, imshow(g) % Fig. 11.14(d).
```

As Fig. 11.14(d) shows, the segmentation using Otsu's method was effective. The borders of the polymersome cells were extracted from the background with reasonable accuracy, despite the relatively low value of the separability measure. ■

All the parameters of the between-class variance are based on the image histogram. As you will see shortly, there are applications in which it is advantageous to be able to compute Ousu's threshold using the histogram, rather than the image, as in function `graythresh`. The following custom function computes `T` and `SM` given the image histogram.

otsuthresh

```
function [T, SM] = otsuthresh(h)
%OTSUTHRESH Otsu's optimum threshold given a histogram.
%   [T, SM] = OTSUTHRESH(H) computes an optimum threshold, T, in the
%   range [0 1] using Otsu's method for a given a histogram, H.

% Normalize the histogram to unit area. If h is already normalized,
% the following operation has no effect.
h = h/sum(h);
h = h(:); % h must be a column vector for processing below.

% All the possible intensities represented in the histogram (256 for
% 8 bits). (i must be a column vector for processing below.)
i = (1:numel(h))';

% Values of P1 for all values of k.
P1 = cumsum(h);

% Values of the mean for all values of k.
m = cumsum(i.*h);

% The image mean.
mG = m(end);
```

```
% The between-class variance.
sigSquared = ((mG*P1 - m).^2)./(P1.*(1 - P1) + eps);

% Find the maximum of sigSquared. The index where the max occurs is
% the optimum threshold. There may be several contiguous max values.
% Average them to obtain the final threshold.
maxSigsq = max(sigSquared);
T = mean(find(sigSquared == maxSigsq));

% Normalized to range [0 1]. 1 is subtracted because MATLAB indexing
% starts at 1, but image intensities start at 0.
T = (T - 1)/(numel(h) - 1);

% Separability measure.
SM = maxSigsq / (sum(((i - mG).^2) .* h) + eps);
```

It is easily verified that this function gives identical results to `graythresh`.

11.3.4 Using Image Smoothing to Improve Global Thresholding

Noise can turn a simple thresholding problem into an unsolvable one. When noise cannot be reduced at the source, and thresholding is the segmentation method of choice, a technique that often enhances performance is to smooth the image prior to thresholding. We introduce the approach using an example.

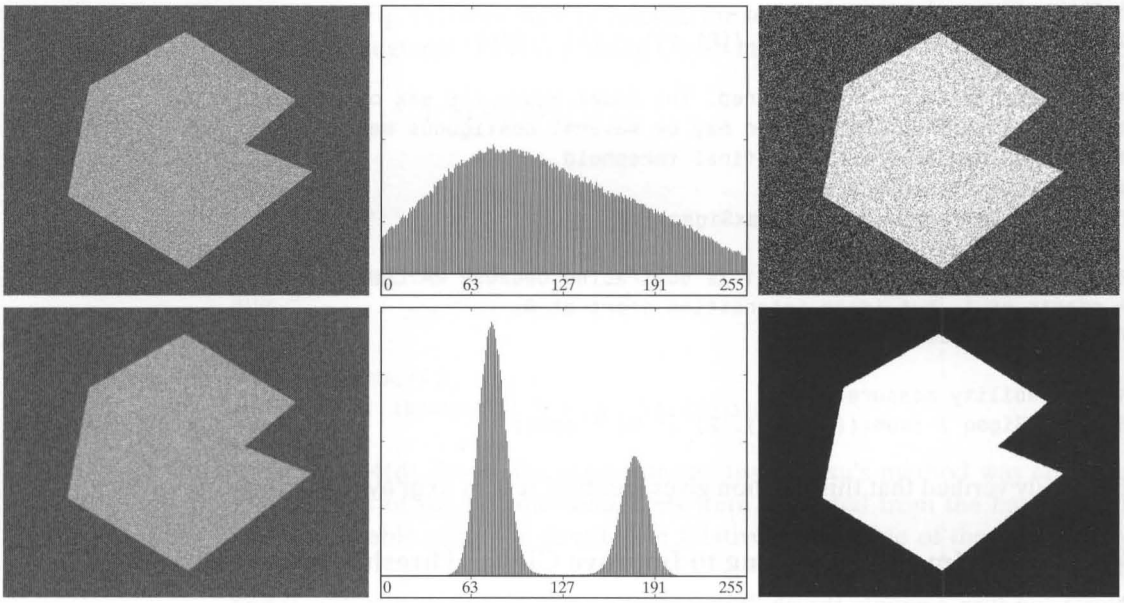
In the absence of noise, the original of Fig. 11.15(a) is bivalued, and can be thresholded perfectly using any threshold placed between the two image intensity values. The image in Fig. 11.15(a) is the result of adding to the original bivalued image Gaussian noise with zero mean and a standard deviation of 50 intensity levels. The histogram of the noisy image [Fig. 11.15(b)] indicates clearly that thresholding is likely to fail on the image as is. The result in Fig. 11.15(c), obtained using Otsu's method, confirms this (every dark point on the object and every light point on the background is a thresholding error, so the segmentation was highly unsuccessful).

Figure 11.15(d) shows the result of smoothing the noisy image with an averaging mask of size 5×5 (the image is of size 651×814 pixels), and Fig. 11.15(e) is its histogram. The improvement in the shape of the histogram due to smoothing is evident, and we would expect thresholding of the smoothed image to be nearly perfect. As Fig. 11.15(f) shows, this indeed was the case. The slight distortion of the boundary between object and background in the segmented, smoothed image was caused by the blurring of the boundary. In fact, the more aggressively we smooth an image the more boundary errors we should anticipate in the segmented result.

The images in Fig. 11.15 were generated using the following commands:

```
>> f = imread('septagon.tif');
```

To obtain Fig. 11.15(a) we added Gaussian noise of zero mean and standard deviation of 50 intensity levels to this image using function `imnoise`. The



a b c
d e f

FIGURE 11.15 (a) Noisy image, and (b) its histogram. (c) Result obtained using Otsu's method. (d) Noisy image smoothed using a 5×5 averaging mask, and (e) its histogram. (f) Result of thresholding using Otsu's method.

toolbox uses variance as an input and it assumes that the intensity range is [0, 1]. Because we are using 255 levels, the variance input into `imnoise` was $50^2/255^2 = 0.038$:

```
>> fn = imnoise(f,'gaussian', 0, 0.038);
>> imshow(fn) % Fig. 11.15(a).
```

The rest of the images in Fig. 11.15 were generated as follows:

```
>> figure, imhist(fn) % Fig. 11.15(b);
>> Tn = graythresh(fn);
>> gn = im2bw(fn, Tn);
>> figure, imshow(gn)
>> % Smooth the image and repeat.
>> w = fspecial('average', 5);
>> fa = imfilter(fn, w, 'replicate');
>> figure, imshow(fa) % Fig. 11.15(d).
>> figure, imhist(fa) % Fig. 11.15(e).
>> Ta = graythresh(fa);
>> ga = im2bw(fa, Ta);
>> figure, imshow(ga) % Fig. 11.15(f).
```

11.3.5 Using Edges to Improve Global Thresholding

Based on the discussion in the previous four sections, we conclude that the chances of selecting a "good" threshold are enhanced considerably if the histogram peaks are tall, narrow, symmetric, and separated by deep valleys. One approach for improving the shape of histograms is to consider only those pixels that lie on or near the edges between objects and the background. An immediate and obvious improvement is that histograms would be less dependent on the relative sizes of objects and the background. In addition, the probability that any of those pixels lies on an object would be approximately equal to the probability that it lies on the background, thus improving the symmetry of the histogram peaks. Finally, as indicated in the following paragraph, using pixels that satisfy some simple measures based on the gradient has a tendency to deepen the valley between histogram peaks.

The approach just discussed assumes that the edges between objects and background are known. This information clearly is not available during segmentation, as finding a division between objects and background is precisely what segmentation is all about. However, an *indication* of whether a pixel is on an edge may be obtained by computing its gradient or the absolute value of the Laplacian (remember, the Laplacian of an image has both positive and negative values). Typically, comparable results are obtained using either method.

The preceding discussion is summarized in the following algorithm, where $f(x, y)$ is the input image:

1. Compute an edge image from $f(x, y)$ using any of the methods discussed in Section 11.1. The edge image can be the gradient or the absolute value of the Laplacian.
2. Specify a threshold value, T.
3. Threshold the image from step 1 using the threshold from step 2 to produce a binary image, $g_T(x, y)$. This image is used as a *marker image* in step 4 to select pixels from $f(x, y)$ corresponding to "strong" edge pixels.
4. Compute a histogram using only the pixels in $f(x, y)$ that correspond to the locations of the 1-valued pixels in $g_T(x, y)$.
5. Use the histogram from step 4 to segment $f(x, y)$ globally using, for example, Otsu's method.

It is customary to specify the value of T corresponding to a percentile,[†] which typically is set high (e.g., in the high 90's) so that few pixels in the edge image are used in the computation of the threshold. Custom function `percentile2i` (see Appendix C) can be used for this purpose. The function computes an intensity value, `I`, corresponding to a specified percentile, `P`. Its syntax is

```
I = percentile2i(h, P)
```

`percentile2i`

See also function `i2percentile` (Appendix C), which computes a percentile given an intensity value.

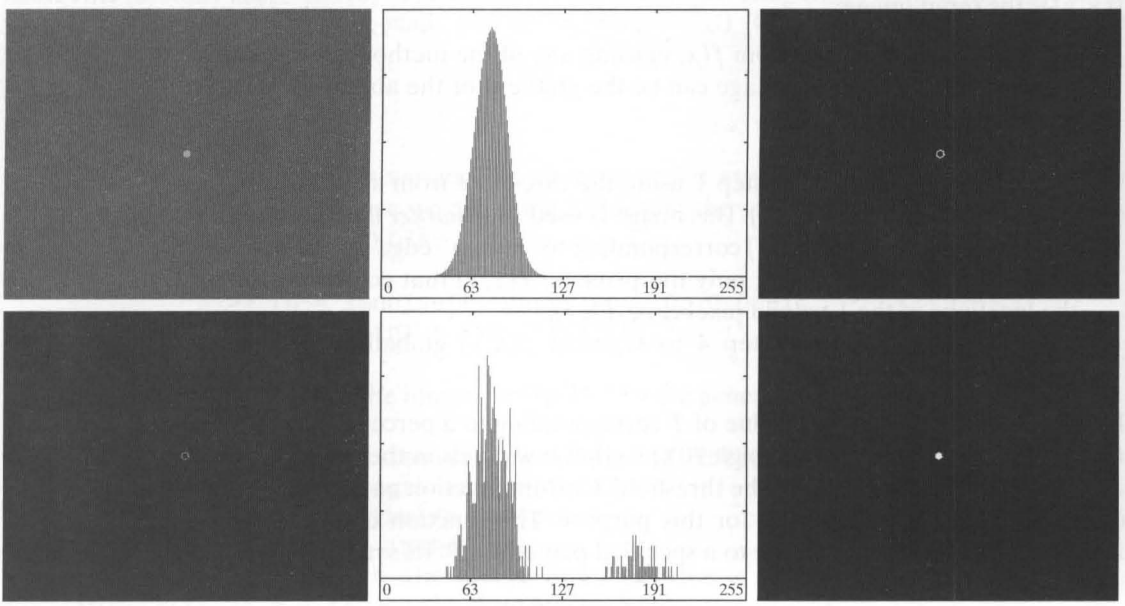
[†] The nth percentile is the smallest number that is greater than n% of the numbers in a given set. For example, if you received a 95 in a test and this score was greater than 80% of all the students taking the test, then you would be in the 80th percentile with respect to the test scores. We define the lowest number in the set to be the 0th percentile and the highest to be the 100th percentile.

where `h` is the image histogram and `P` is a percentile value in the range [0, 1]. Output `I` is the intensity level (also in the range [0, 1]) corresponding to the `P`th percentile.

EXAMPLE 11.9: Using edge information based on the gradient to improve global thresholding.

■ Figure 11.16(a) shows the septagon image severely scaled down in size to a few pixels. The image was corrupted by Gaussian noise with zero mean and a standard deviation of 10 intensity levels. From the histogram in Fig. 11.16(b), which is unimodal, and from our negative experience with a much larger version of the object, we conclude that global thresholding will fail in this case. When objects are much smaller than the background, their contribution to the histogram is negligible. Using edge information can improve the situation. Figure 11.16(c) is the gradient image, obtained as follows:

```
>> f = tofloat(imread('Fig1116(a).tif'));
>> sx = fspecial('sobel');
>> sy = sx';
>> gx = imfilter(f,sx,'replicate');
>> gy = imfilter(f,sy,'replicate');
>> grad = sqrt(gx.*gx + gy.*gy);
>> grad = grad/max(grad(:));
```



a b c
d e f

FIGURE 11.16 (a) Noisy image of small septagon, and (b) its histogram. (c) Gradient magnitude image thresholded at the 99.9 percentile level. (d) Image formed by the product of (a) and (c). (e) Histogram of the nonzero pixels in the image in (d). (f) Result of segmenting image (a) with the Otsu threshold found using the histogram in (e). (The threshold found was 133.5, which is approximately midway between the peaks in this histogram.)

where the last command normalizes the values of `grad` to the correct [0, 1] range for a floating point image. Next, we obtain the histogram of `grad` and use it to estimate the threshold for the gradient, using a high (99.9) percentile (remember, we want to keep only the large values of the gradient image, which should occur near the borders of the object and the background:

```
>> h = imhist(grad);
>> Q = percentile2i(h, 0.999);
```

where `Q` is in the range [0, 1]. The next steps are: threshold the gradient using `Q`, form the marker image and use it to extract from `f` the points at which the gradient values are greater than `Q`, and obtain the histogram of the result:

```
>> markerImage = grad > Q;
>> figure, imshow(markerImage) % Fig. 11.16(c).
>> fp = f.*markerImage;
>> figure, imshow(fp) % Fig. 11.16(d).
>> hp = imhist(fp);
```

Image `fp` contains the pixels of `f` around the border of the object and background. Thus its histogram is dominated by 0s. Because we are interested in segmenting the values around the border of the object, we need to eliminate the contribution of the 0s to the histogram, so we exclude the first element of `hp`, and then use the resulting histogram to obtain the Otsu threshold:

```
>> hp(1) = 0;
>> bar(hp, 0) % Fig. 11.16(e).
>> T = otsuthresh(hp);
>> T*(numel(hp) - 1)
ans =
   133.5000
```

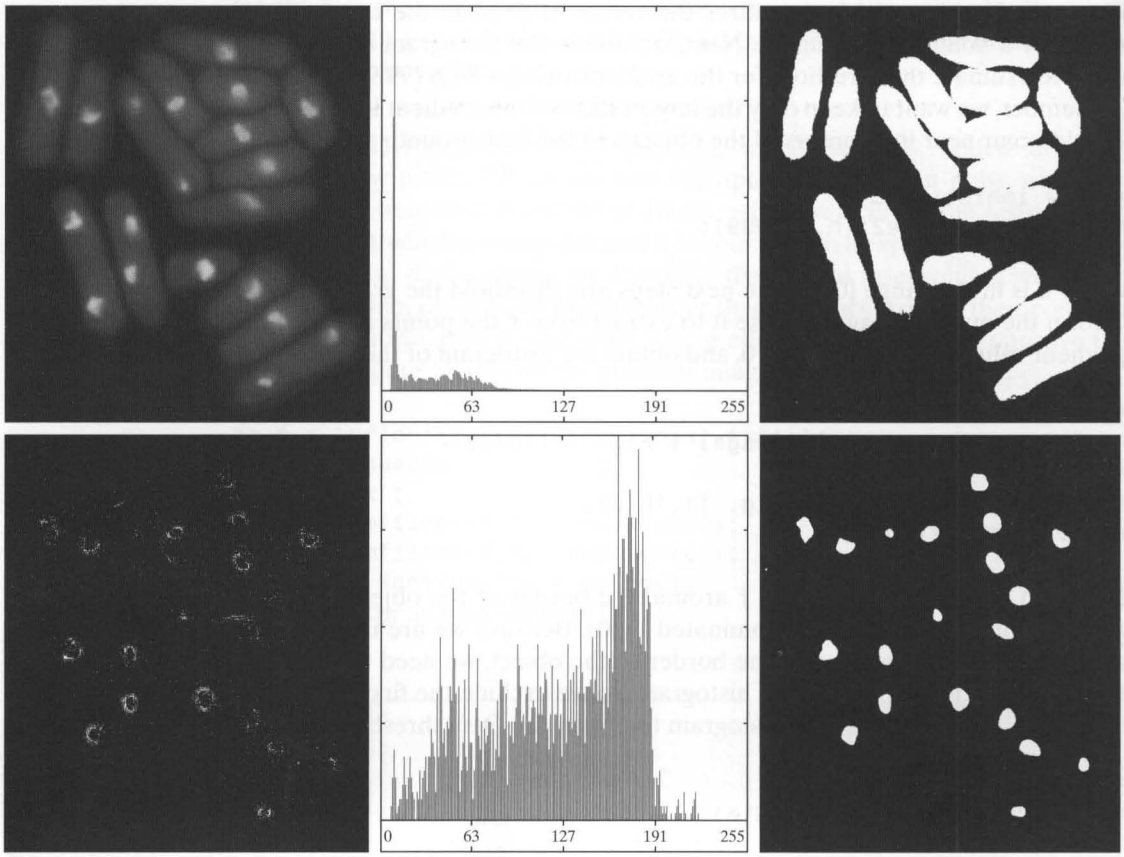
Histogram `hp` is shown in Fig. 11.16(e). Observe that now we have distinct, relatively narrow peaks separated by a deep valley, as desired, and the optimum threshold is near the mid point between the modes. Thus, we expect a nearly perfect segmentation:

```
>> g = im2bw(f, T);
>> figure, imshow(g) % Fig. 11.16(f).
```

As Fig. 11.16(f) shows, the image was indeed segmented properly. ■

EXAMPLE 11.10: Using Laplacian edge information to improve global thresholding.

■ In this example we consider a more complex thresholding problem, and illustrate how to use the Laplacian to obtain edge information that leads to improved segmentation. Figure 11.17(a) is an 8-bit image of yeast cells in which we wish to use global thresholding to obtain the regions corresponding to the bright spots. As a starting point, Fig. 11.17(b) shows the image histogram, and



a b c
d e f

FIGURE 11.17 (a) Image of yeast cells. (b) Histogram of (a). (c) Segmentation of (a) using function `graythresh`. (d) Product of the marker and original images. (e) Histogram of the nonzero pixels in (d). (f) Image thresholded using Otsu's method based on the histogram in (e). (Original image courtesy of Professor Susan L. Forsburg, University of Southern California.)

Fig. 11.17(c) is the result obtained using Otsu's method directly on the image:

```
>> f = tofloat(imread('Fig1117(a).tif'));
>> imhist(f) % Fig. 11.17(b).
>> hf = imhist(f);
>> [Tf SMf] = graythresh(f);
>> gf = im2bw(f, Tf);
>> figure, imshow(gf) % Fig. 11.17(c).
```

We see that Otsu's method failed to achieve the original objective of detecting the bright spots and, while the method was able to isolate some of the cell

regions themselves, several of the segmented regions on the right are not disjoint. The threshold computed by the Otsu method was 42 and the separability measure was 0.636. The following steps are similar to those in Example 11.9, with the exception that we use the absolute value of the Laplacian to obtain edge information, and we used a slightly lower percentile because the histogram of the thresholded Laplacian was more sparse than in the previous example:

```
>> w = [-1 -1 -1; -1 8 -1; -1 -1 -1];
>> lap = abs(imfilter(f, w, 'replicate'));
>> lap = lap/max(lap(:));
>> h = imhist(lap);
>> Q = percentile2i(h, 0.995);
>> markerImage = lap > Q;
>> fp = f.*markerImage;
>> figure, imshow(fp) % Fig. 11.17(d).
>> hp = imhist(fp);
>> hp(1) = 0;
>> figure, bar(hp, 0) % Fig. 11.17(e).
>> T = otsuthresh(hp);
>> g = im2bw(f, T);
>> figure, imshow(g) % Fig. 11.17(f).
```

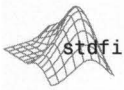
Figure 11.17(d) shows the product of `f` and `markerImage`. Note in this image how the points cluster near the edges of the bright spots, as expected from the preceding discussion. Figure 11.17(e) is the histogram of the nonzero pixels in (d). Finally, Fig. 11.17(f) shows the result of globally segmenting the original image using Otsu's method based on the histogram in Fig. 11.17(e). This result agrees with the locations of the bright spots in the image. The threshold computed by the Otsu method was 115 and the separability measure was 0.762, both of which are higher than the values obtained directly from the image. ■

11.3.6 Variable Thresholding Based on Local Statistics

Global thresholding methods typically fail when the background illumination is highly nonuniform. One solution to this problem is to attempt to estimate the shading function, use it to compensate for the nonuniform intensity pattern, and then threshold the image globally using one of the methods discussed above. You saw an example of this approach in Section 10.6.2. Another approach used to compensate for irregularities in illumination, or in cases where there is more than one dominant object intensity (in which case global thresholding also has difficulties), is to use variable thresholding. This approach computes a threshold value at every point (x, y) in the image, based on one or more specified properties of the pixels in a neighborhood of (x, y).

We illustrate the basic approach to local thresholding using the standard deviation and mean of the pixels in a neighborhood of every point in an image. These two quantities are quite useful for determining local thresholds because they are descriptors of local contrast and average intensity. Let σ_{xy} and m_{xy}

denote the standard deviation and mean value of the set of pixels contained in a neighborhood that is centered at coordinate (x, y) in an image. To compute the local standard deviation, we use function `stdfilt`, which has the following syntax:




```
g = stdfilt(f, nhood)
```

where `f` is the input image and `nhood` is an array of zeros and ones in which the nonzero elements specify the neighbors used in the computation of the local standard deviation. The size of `nhood` must be odd in each dimension; the default value is `ones(3)`.

To compute the local means, we use the following custom function:

localmean

```
function mean = localmean(f, nhood)
%LOCALMEAN Computes an array of local means.
%  MEAN = LOCALMEAN(F, NHOOD) computes the mean at the center of
%  every neighborhood of F defined by NHOOD, an array of zeros and
%  ones where the nonzero elements specify the neighbors used in the
%  computation of the local means. The size of NHOOD must be odd in
%  each dimension; the default is ones(3). Output MEAN is an array
%  the same size as F containing the local mean at each point.

if nargin == 1
   nhood = ones(3) / 9;
else
   nhood = nhood / sum(nhood(:));
end
mean = imfilter(tofloat(f), nhood, 'replicate');
```

The following are common forms of variable, local thresholds based on the local mean and standard deviations:

$$T_{xy} = a\sigma_{xy} + bm_{xy}$$

where a and b are nonnegative constants. Another useful form is

$$T_{xy} = a\sigma_{xy} + bm_G$$

where m_G is the global image mean. The segmented image is computed as

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) > T_{xy} \\ 0 & \text{if } f(x,y) \leq T_{xy} \end{cases}$$

where $f(x, y)$ is the input image. This equation is evaluated and applied at all pixel locations.

Significant power can be added to local thresholding by combining local properties logically instead of arithmetically, as above. For example. we can define local thresholding in terms of a logical AND as follows:

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) > a\sigma_{xy} \text{ AND } f(x,y) > bm \\ 0 & \text{otherwise} \end{cases}$$

where m is either the local mean, m_{xy}, or the global mean, m_G, as defined above. The following function implements local thresholding using this formulation. The basic structure of this function can be adapted easily to other combinations of logical and/or local operations.

localthresh

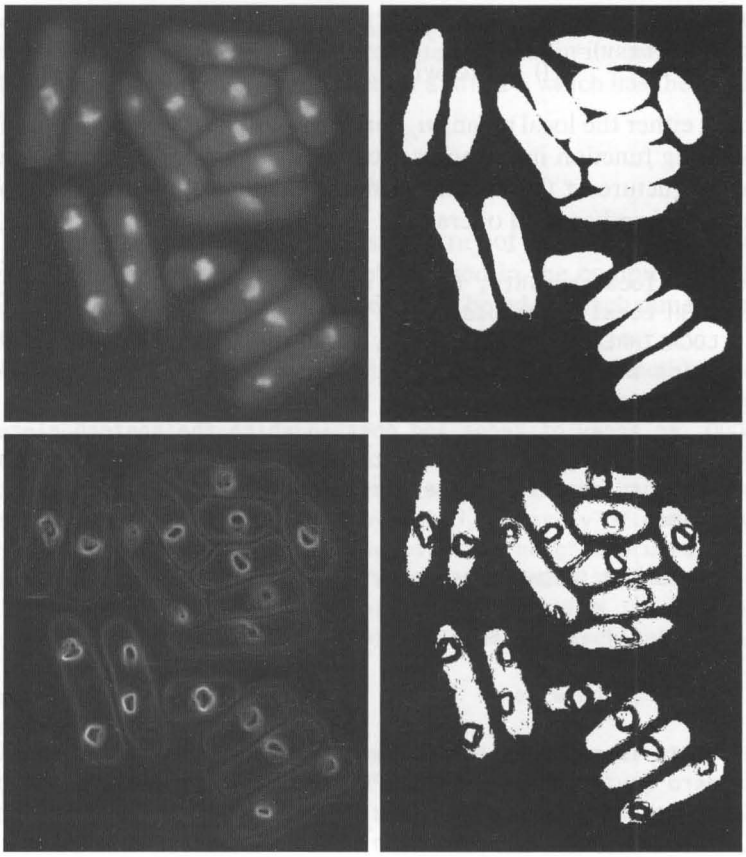
```
function g = localthresh(f, nhood, a, b, meantype)
%LOCALTHRESH Local thresholding.
%   G = LOCALTHRESH(F, NHOOD, A, B, MEANTYPE) thresholds image F by
%   computing a local threshold at the center,(x, y), of every
%   neighborhood in F. The size of the neighborhoods is defined by
%   NHOOD, an array of zeros and ones in which the nonzero elements
%   specify the neighbors used in the computation of the local mean
%   and standard deviation. The size of NHOOD must be odd in both
%   dimensions.
%
%   The segmented image is given by
%
%           1  if (F > A*SIG) AND (F > B*MEAN)
%       G =
%           0  otherwise
%
%   where SIG is an array of the same size as F containing the local
%   standard deviations. If MEANTYPE = 'local' (the default), then
%   MEAN is an array of local means.  If MEANTYPE = 'global', then
%   MEAN is the global (image) mean, a scalar.  Constants A and B
%   are nonnegative scalars.

% Intialize.
f = tofloat(f);

% Compute the local standard deviations.
SIG = stdfilt(f, nhood);
% Compute MEAN.
if nargin == 5 && strcmp(meantype,'global')
   MEAN = mean2(f);
else
   MEAN = localmean(f, nhood); % This is a custom function.
end

% Obtain the segmented image.
g = (f > a*SIG) & (f > b*MEAN);
```

EXAMPLE 11.11: Comparing global and local thresholding.

■ Figure 11.18(a) shows the image from Example 11.10. We want to segment the cells from the background, and the nuclei (inner, brighter regions) from the body of the cells. This image has three predominant intensity levels, so it is reasonable to expect that such a segmentation is possible. However, it is

a b
c d

FIGURE 11.18
(a) Yeast cell image. (b) Image segmented using Otsu's method. (c) Image of local standard deviations. (d) Image segmented using local thresholding.



highly unlikely that a single global threshold can do the job; this is verified in Fig. 11.18(b), which shows the result of using Otsu's method:

```
>> [TGlobal] = graythresh(f);
>> gGlobal = im2bw(f, TGlobal);
>> imshow(gGlobal) % Fig. 11.18(b).
```

where `f` is the image in Fig. 11.18(a). As the figure shows, it was possible to partially segment the cells from the background (some segmented cells are joined) but the method could not extract the cell nuclei.

Because the nuclei are significantly brighter than the bodies of the cells, we would expect the local standard deviations to be relatively large around the borders of the nuclei and somewhat less around the borders of the cells. As Fig. 11.18(c) shows, this indeed is the case. Thus, we conclude that the predicate in function `localthresh`, which is based on local standard deviations, should be helpful:

```
>> g = localthresh(f, ones(3), 30, 1.5, 'global');
>> SIG = stdfilt(f, ones(3));
```

```
>> figure, imshow(SIG, [ ]) % Fig. 11.18(c).
>> figure, imshow(g) % Fig. 11.18(d).
```

As Fig. 11.18(d) shows, the segmentation using a predicate was quite effective. The cells were segmented individually from the background, and all the nuclei were segmented properly. The values used in the function were determined experimentally, as is usually the case in applications such as this. Choosing the global mean generally gives better results when the background is nearly constant and all the object intensities are above or below the background intensity. ■

11.3.7 Image Thresholding Using Moving Averages

A special case of the local thresholding method discussed in the previous section is based on computing a moving average along scan lines of an image. This implementation is quite useful in document processing, where speed is a fundamental requirement. The scanning typically is carried out line by line in a zigzag pattern to reduce illumination bias. Let z_{k+1} denote the intensity of the point encountered in the scanning sequence at step $k+1$. The *moving average* (mean intensity) at this new point is given by

$$m(k+1) = \frac{1}{n}\sum_{i=k+2-n}^{k+1} z_i$$

$$= m(k) + \frac{1}{n}(z_{k+1} - z_{k-n})$$

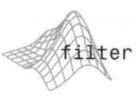
The first line of this equation is valid for $k \geq n-1$. When k is less than $n-1$, averages are formed using the available points. Similarly, the second line is valid for $k \geq n+1$.

where n denotes the number of points used in computing the average and $m(1) = z_1/n$. This initial value is not strictly correct because the average of a single point is the value of the point itself. However, we use $m(1) = z_1/n$ so that no special computations are required when the preceding averaging equation first starts up. Another way of viewing it is that this is the value we would obtain if the border of the image were padded with $n-1$ zeros. The algorithm is initialized only once, not at every row. Because a moving average is computed for every point in the image, segmentation is implemented using

$$f(x,y) = \begin{cases} 1 & \text{if } f(x,y) > Km_{xy} \\ 0 & \text{otherwise} \end{cases}$$

where K is constant in the range $[0, 1]$, and m_{xy} is the moving average at point (x, y) in the input image.

The following custom function implements the concepts just discussed. The function uses MATLAB function `filter`, a 1-D filtering function with the basic syntax




```
Y = filter(c, d, X)
```

This function filters the data in vector `X` with the filter described by numerator coefficient vector `c` and denominator coefficient vector `d`. If `d = 1` (a scalar) the coefficients in `c` define the filter completely.

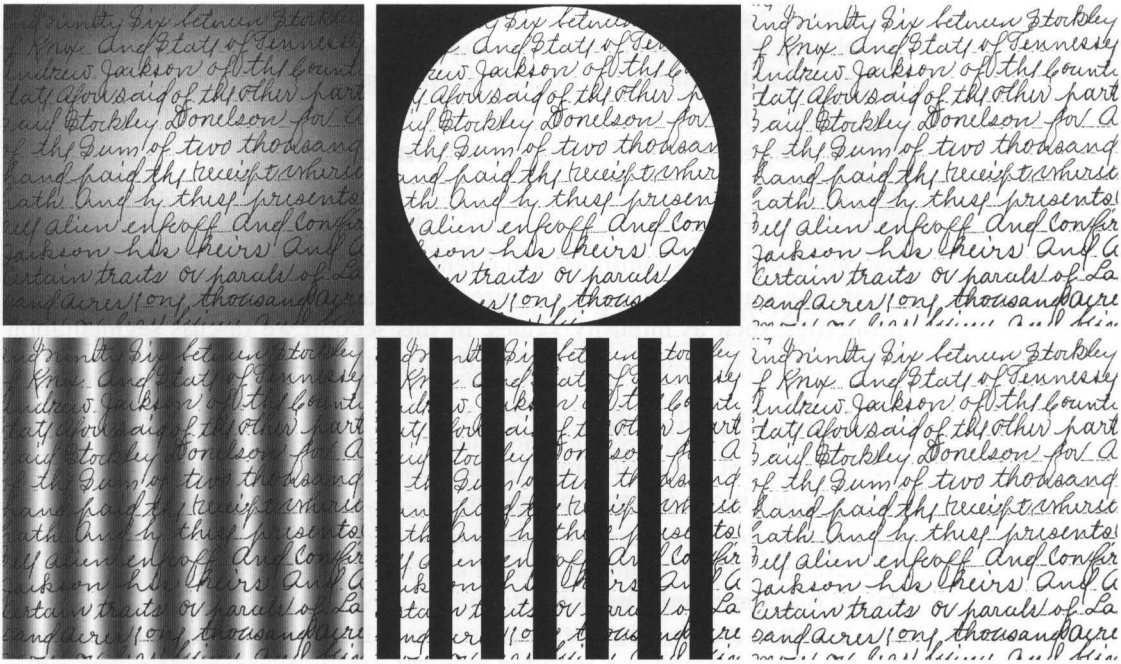
movingthresh

```
function g = movingthresh(f, n, K)
%MOVINGTHRESH Image segmentation using a moving average threshold.
%   G = MOVINGTHRESH(F, n, K) segments image F by thresholding its
%   intensities based on the moving average of the intensities along
%   individual rows of the image. The average at pixel k is formed
%   by averaging the intensities of that pixel and its n - 1
%   preceding neighbors. To reduce shading bias, the scanning is
%   done in a zig-zag manner, treating the pixels as if they were a
%   1-D, continuous stream. If the value of the image at a point
%   exceeds K percent of the value of the running average at that
%   point, a 1 is output in that location in G. Otherwise a 0 is
%   output. At the end of the procedure, G is thus the thresholded
%   (segmented) image. K must be a scalar in the range [0, 1].

% Preliminaries.
f = tofloat(f);
[M, N] = size(f);
if (n < 1) || (rem(n, 1) ~= 0)
   error('n must be an integer >= 1.')
end
if K < 0 || K > 1
   error('K must be a fraction in the range [0, 1].')
end

% Flip every other row of f to produce the equivalent of a zig-zag
% scanning pattern. Convert image to a vector.
f(2:2:end, :) = fliplr(f(2:2:end, :));
f = f'; % Still a matrix.
f = f(:)'; % Convert to row vector for use in function filter.

% Compute the moving average.
maf = ones(1, n)/n; % The 1-D moving average filter.
ma = filter(maf, 1, f); % Computation of moving average.

% Perform thresholding.
g = f > K * ma;

% Go back to image format (indexed subscripts).
g = reshape(g, N, M)';
% Flip alternate rows back.
g(2:2:end, :) = fliplr(g(2:2:end, :));
```

EXAMPLE 11.12: Image thresholding using moving averages.

■ Figure 11.19(a) shows an image of handwritten text shaded by a spot intensity pattern. This form of intensity shading can occur, for example, in images obtained with a photographic flash. Figure 11.19(b) is the result of segmentation using the Otsu global thresholding method:

```
>> f = imread('Fig1119(a).tif');
>> T = graythresh(f);
>> g1 = im2bw(f, T); % Fig. 11.19(b).
```



a b c
d e f

FIGURE 11.19 (a) Text image corrupted by spot shading. (b) Result of global thresholding using Otsu's method. (c) Result of local thresholding using moving averages. (d)-(f) Results of using the same sequence of operations on an image corrupted by sinusoidal shading.

It is not unexpected that global thresholding could not overcome the intensity variation. Figure 11.19(c) shows successful segmentation with local thresholding using moving averages:

```
>> g2 = movingthresh(f, 20, 0.5);
>> figure, imshow(g2) % Fig. 11.19(c).
```

A rule of thumb is to let the width of the averaging window be five times the average stroke width. In this case, the average width was 4 pixels, so we let $n = 20$ and used $K = 0.5$ (the algorithm is not particularly sensitive to the values of these parameters).

As another illustration of the effectiveness of this segmentation approach, we used the same parameters as in the previous paragraph to segment the image in Fig. 11.19(d), which is corrupted by a sinusoidal intensity variation typical of the variations that may occur when the power supply in a document scanner is not grounded properly. As Figs. 11.19(e) and (f) show, the segmentation results are similar to those in the first row of Fig. 11.19.

Observe that successful segmentation results were obtained in both cases using the same values for n and K, illustrating the relative ruggedness of the

approach. In general, thresholding based on moving averages works well when the objects of interest are small (or thin) with respect to the image size, a condition generally satisfied by images of typed or handwritten text. ■

11.4 Region-Based Segmentation

The objective of segmentation is to partition an image into regions. In Sections 11.1 and 11.2 we approached this problem by finding boundaries between regions based on discontinuities in intensity levels, whereas in Section 11.3 segmentation was accomplished via thresholds based on the distribution of pixel properties, such as intensity values. In this section we discuss segmentation techniques that are based on finding the regions directly.

11.4.1 Basic Formulation

Let R represent the entire image region. We may view segmentation as a process that partitions R into n subregions, $R_1, R_2, \ldots, R_n$, such that

(a) $\bigcup_{i=1}^{n} R_i = R$.

(b) R_i is a connected region, $i = 1, 2, \ldots, n$.

(c) $R_i \cap R_j = \varnothing$ for all i and j, $i \neq j$.

(d) $P(R_i) = \text{TRUE}$ for $i = 1, 2, \ldots, n$.

(e) $P(R_i \cup R_j) = \text{FALSE}$ for any adjacent regions R_i and R_j.

In the context of the discussion in Section 10.4, two disjoint regions, R_i and R_j, are said to be *adjacent* if their union forms a connected component.

Here, $P(R_i)$ is a *logical predicate* defined over the points in set R_i and $\varnothing$ is the null set.

Condition (a) indicates that the segmentation must be complete; that is, every pixel must be in a region. The second condition requires that points in a region be connected (e.g., 4- or 8-connected). Condition (c) indicates that the regions must be disjoint. Condition (d) deals with the properties that must be satisfied by the pixels in a segmented region—for example, "$P(R_i) = \text{TRUE}$ if all pixels in R_i have the same intensity level." Finally, condition (e) indicates that adjacent regions R_i and R_j are different in the sense of predicate P.

11.4.2 Region Growing

As its name implies, *region growing* is a procedure that groups pixels or subregions into larger regions based on predefined criteria for growth. The basic approach is to start with a set of "seed" points and from these grow regions by appending to each seed those neighboring pixels that have predefined properties similar to the seed (such as specific ranges of gray level or color).

Selecting a set of one or more seed points often can be based on the nature of the problem, as we show later in Example 11.14. When a priori information is not available, one procedure is to compute at every pixel the same set of properties that ultimately will be used to assign pixels to regions during the growing process. If the result of these computations shows clusters of values, the pixels whose properties place them near the centroid of these clusters can be used as seeds.

The selection of similarity criteria depends not only on the problem under consideration, but also on the type of image data available. For example, the analysis of land-use satellite imagery depends heavily on the use of color. This problem would be significantly more difficult, or even impossible, to handle without the inherent information available in color images. When the images are monochrome, region analysis must be carried out with a set of descriptors based on intensity levels (such as moments or texture) and spatial properties (such as connectivity). We discuss descriptors useful for region characterization in Chapter 12.

Descriptors alone can yield misleading results if connectivity (adjacency) information is not used in the region-growing process. For example, visualize a random arrangement of pixels with only three distinct intensity values. Grouping pixels with the same intensity level to form a "region" without taking connectivity into consideration would yield a segmentation result that is meaningless in the context of this discussion.

Another problem in region growing is the formulation of a stopping rule. Basically, growing a region should stop when no more pixels satisfy the criteria for inclusion in that region. Criteria such as intensity values, texture, and color, are local in nature and do not take into account the "history" of region growth. Additional criteria that increase the power of a region-growing algorithm utilize the concept of size, likeness between a candidate pixel and the pixels grown so far (such as a comparison of the intensity of a candidate and the average intensity of the grown region), and the shape of the region being grown. The use of these types of descriptors is based on the assumption that a model of expected results is at least partially available.

To illustrate the principles of how region segmentation can be handled in MATLAB, we develop next an M-function, called `regiongrow`, to do basic region growing. The syntax for this function is

```
[g, NR, SI, TI] = regiongrow(f, S, T)
```

where `f` is an image to be segmented and parameter `S` can be an array (the same size as `f`) or a scalar. If `S` is an array, it must contain 1s at all the coordinates where seed points are located and 0s elsewhere. Such an array can be determined by inspection, or by an external seed-finding function. If `S` is a scalar, it defines an intensity value such that all the points in `f` with that value become seed points. Similarly, `T` can be an array (the same size as `f`) or a scalar. If `T` is an array, it contains a threshold value for each location in `f`. If `T` is a scalar, it defines a global threshold. The threshold value(s) is (are) used to test if a pixel in the image is sufficiently similar to the seed or seeds to which it is 8-connected. All values of `S` and `T` must be scaled to the range [0, 1], independently of the class of the input image.

For example, if `S = a` and `T = b`, and we are comparing intensities, then a pixel is said to be similar to `a` (in the sense of passing the threshold test) if the absolute value of the difference between its intensity and `a` is less than or equal to `b`. If, in addition, the pixel in question is 8-connected to one or more seed

values, then the pixel is considered a member of one or more regions. Similar comments hold if S and T are arrays, the difference being that comparisons are done between corresponding elements from S and T.

In the output, g is the segmented image, with the members of each region being labeled with a different integer value. Parameter NR is the number of regions found. Parameter SI is an image containing the seed points, and parameter TI is an image containing the pixels that passed the threshold test before they were processed for connectivity. Both SI and TI are of the same size as f.

The code for function regiongrow follows. Note the use of Chapter 10 function bwmorph to reduce to 1 the number of connected seed points in each region in S (when S is an array) and function imreconstruct to find pixels connected to each seed.

regiongrow

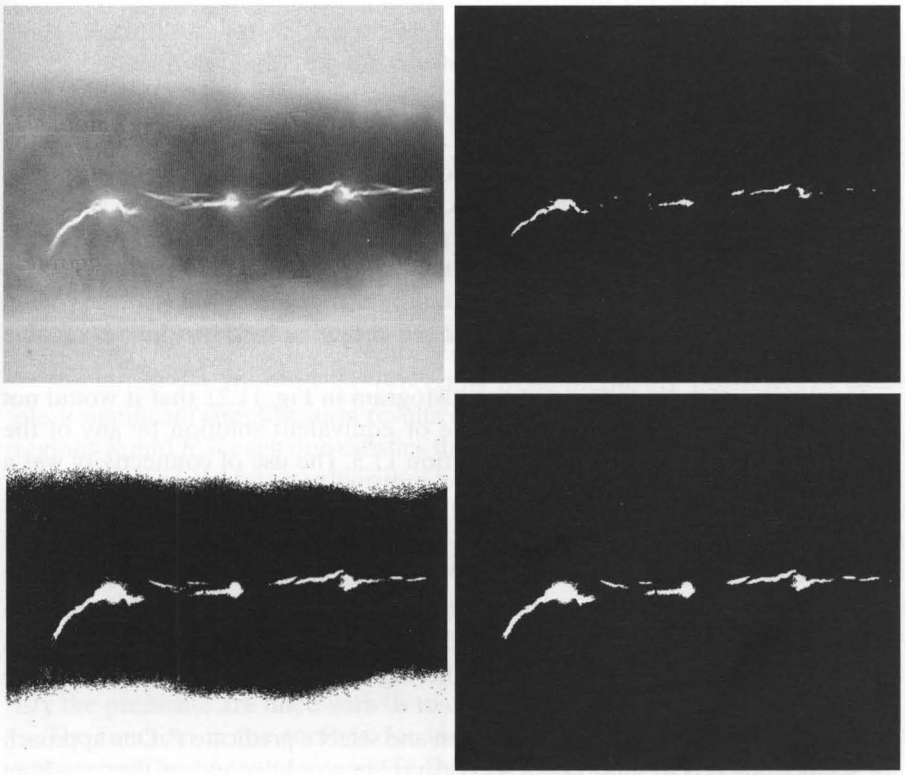
```
function [g, NR, SI, TI] = regiongrow(f, S, T)
%REGIONGROW Perform segmentation by region growing.
%   [G, NR, SI, TI] = REGIONGROW(F, S, T). S can be an array (the
%   same size as F) with a 1 at the coordinates of every seed point
%   and 0s elsewhere.  S can also be a single seed value. Similarly,
%   T can be an array (the same size as F) containing a threshold
%   value for each pixel in F. T can also be a scalar, in which case
%   it becomes a global threshold. All values in S and T must be in
%   the range [0, 1]
%
%   G is the result of region growing, with each region labeled by a
%   different integer, NR is the number of regions, SI is the final
%   seed image used by the algorithm, and TI is the image consisting
%   of the pixels in F that satisfied the threshold test, but before
%   they were processed for connectivity.

f = tofloat(f);
% If S is a scalar, obtain the seed image.
if numel(S) == 1
   SI = f == S;
   S1 = S;
else
   % S is an array. Eliminate duplicate, connected seed locations
   % to reduce the number of loop executions in the following
   % sections of code.
   SI = bwmorph(S, 'shrink', Inf);
   S1 = f(SI); % Array of seed values.
end

TI = false(size(f));
for K = 1:length(S1)
   seedvalue = S1(K);
   S = abs(f - seedvalue) <= T; % Re-use variable S.
   TI = TI | S;
end
```

```
% Use function imreconstruct with SI as the marker image to
% obtain the regions corresponding to each seed in S. Function
% bwlabel assigns a different integer to each connected region.
[g, NR] = bwlabel(imreconstruct(SI, TI));
```

EXAMPLE 11.13: Using region growing to detect weld porosity.

■ Figure 11.20(a) shows an X-ray image of a weld (the horizontal dark region) containing several cracks and porosities (the bright, white streaks running horizontally through the middle of the image). We wish to use function `regiongrow` to segment the regions corresponding to weld failures. These segmented regions could be used for tasks such as automated inspection, for inclusion in a database of historical studies, and for controlling an automated welding system.

The first task is to specify the initial seed points. In this application, it is known that some pixels in areas of defective welds tend to have the maximum allowable digital value (255 in this case). Based in this information, we let `S = 1` (all values of `S` have to be scaled to the range [0, 1]). The next step is to choose a threshold or threshold array. In this example we used a threshold equal to 65 (0.26 when scaled to the range [0, 1]). This number is from analysis of the histogram in Fig. 11.21 and represents the difference between 255 and the location



a b
c d

FIGURE 11.20 (a) Image showing defective welds. (b) Seed points. (c) Binary image showing all the pixels (in white) that passed the threshold test. (d) Result after all the pixels in (c) were analyzed for 8-connectivity to the seed points. (Original image courtesy of X-TEK Systems, Ltd.)

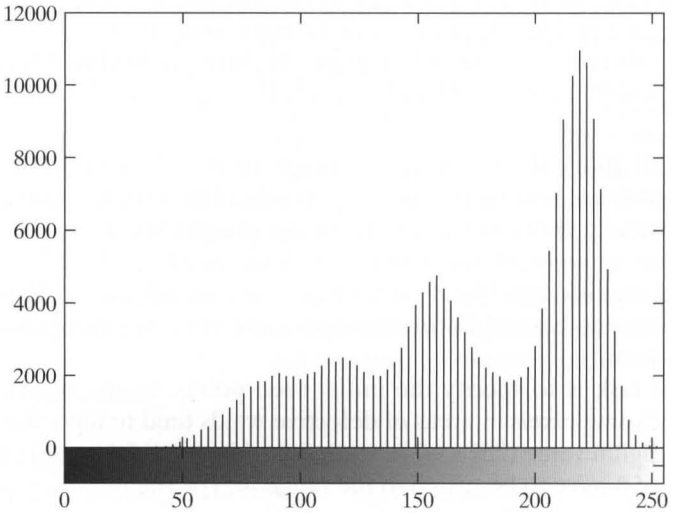


FIGURE 11.21 Histogram of Fig. 11.20(a).

of the first major valley to the left (190), which is representative of the highest intensity value in the dark weld region. The results in Fig. 11.20 were generated with the function call

```
>> [g, NR, SI, TI] = regiongrow(f, 1, 0.26);
```

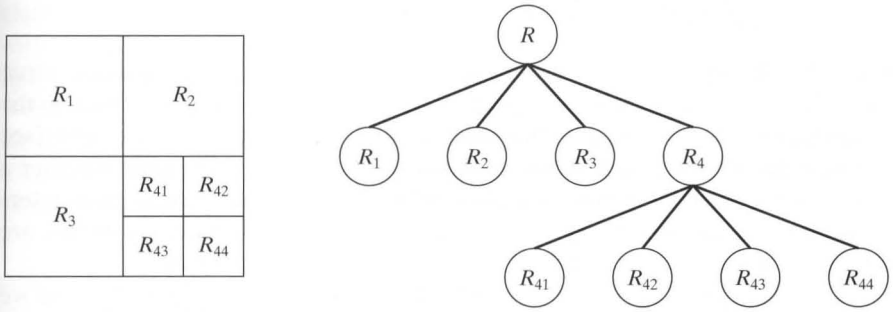
Figure 11.20(b) shows the seed points (image `SI`). They are numerous in this case because the seeds were specified as *all* points in the image with a value of 255 (1 when scaled). Figure 11.20(c) is image `TI`. It shows all the points that passed the threshold test; that is, the points with intensity z_i such that $|z_i - S| \leq T$. Figure 11.20(d) shows the result of extracting all the pixels in Figure 11.20(c) that were connected to the seed points. This is the segmented image, g. It is evident by comparing this image with the original that the region growing procedure did indeed segment the defective welds with a reasonable degree of accuracy.

Finally, we note by looking at the histogram in Fig. 11.21 that it would not have been possible to obtain the same or equivalent solution by any of the thresholding methods discussed in Section 11.3. The use of connectivity was a fundamental requirement in this case. ■

11.4.3 Region Splitting and Merging

The procedure just discussed grows regions from a set of seed points. An alternative is to subdivide an image initially into a set of arbitrary, disjointed regions and then merge and/or split the regions in an attempt to satisfy the conditions stated in Section 11.4.1.

Let R represent the entire image region and select a predicate P. One approach for segmenting R is to subdivide it successively into smaller and smaller quadrant



a b

FIGURE 11.22
(a) Partitioned image.
(b) Corresponding quadtree.

regions so that, for any region R_i, $P(R_i) = \text{TRUE}$. We start with the entire region. If $P(R) = \text{TRUE}$ we divide the image into quadrants. If P is FALSE for any quadrant, we subdivide that quadrant into subquadrants, and so on. This particular splitting technique has a convenient representation in the form of a so-called *quadtree*; that is, a tree in which each node has exactly four descendants, as Fig. 11.22 shows (the subimages corresponding to the nodes of a quadtree sometimes are called *quadregions* or *quadimages*). Note that the root of the tree corresponds to the entire image and that each node corresponds to the subdivision of a node into four descendant nodes. In this case, only R_4 was subdivided further.

If only splitting is used, the final partition normally contains adjacent regions with identical properties. This drawback can be remedied by allowing merging, as well as splitting. Satisfying the constraints of Section 11.4.1 requires merging only adjacent regions whose combined pixels satisfy the predicate P. That is, two adjacent regions R_i and R_j are merged only if $P(R_i \cup R_j) = \text{TRUE}$.

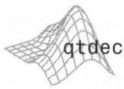
The preceding discussion may be summarized by the following procedure in which, at any step,

1. Split into four disjoint quadrants any region R_i for which $P(R_i) = \text{FALSE}$.
2. When no further splitting is possible, merge any adjacent regions R_i and R_j for which $P(R_i \cup R_j) = \text{TRUE}$.
3. Stop when no further merging is possible.

To keep notation as simple as possible, we let R_i and R_j denote any two regions during splitting and merging. Attempting to introduce notation that reflects various of levels of splitting and/or merging (as in Fig. 11.22) would complicate the explanation unnecessarily.

Numerous variations of the preceding basic theme are possible. For example, a significant simplification results if we allow merging of any two adjacent regions R_i and R_j if each one satisfies the predicate individually. This results in a much simpler (and faster) algorithm because testing of the predicate is limited to individual quadregions. As Example 11.14 later in this section shows, this simplification is still capable of yielding good segmentation results in practice. Using this approach in step 2 of the procedure, all quadregions that satisfy the predicate are filled with 1s and their connectivity can be easily examined using, for example, function `imreconstruct`. This function, in effect, accomplishes the desired merging of adjacent quadregions. The quadregions that do not satisfy the predicate are filled with 0s to create a segmented image.

The function in the toolbox for implementing quadtree decomposition is `qtdecomp`. The syntax of interest in this section is

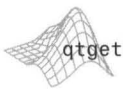



```
Z = qtdecomp(f, @split_test, parameters)
```

Other forms of `qtdecomp` are discussed in Section 12.2.2.

where `f` is the input image and `Z` is a sparse matrix containing the quadtree structure. If `Z(k, m)` is nonzero, then `(k, m)` is the upper-left corner of a block in the decomposition and the size of the block is `Z(k, m)`. Function `split_test` (see function `splitmerge` below for an example) is used to determine whether a region is to be split or not, and `parameters` are any additional parameters (separated by commas) required by `split_test`. The mechanics of this are similar to those discussed in Section 3.4.2 for function `coltfilt`.

To get the actual quadregion pixel values in a quadtree decomposition we use function `qtgetblk`, with syntax




```
[vals, r, c] = qtgetblk(f, Z, m)
```

where `vals` is an array containing the values of the blocks of size `m` × `m` in the quadtree decomposition of `f`, and `Z` is the sparse matrix returned by `qtdecomp`. Parameters `r` and `c` are vectors containing the row and column coordinates of the upper-left corners of the blocks.

We illustrate the use of function `qtdecomp` by writing a basic split-and-merge M-function that uses the simplification discussed earlier, in which two regions are merged if each satisfies the predicate individually. The function, which we call `splitmerge`, has the following calling syntax:

```
g = splitmerge(f, mindim, @predicate)
```

where `f` is the input image and `g` is the output image in which each connected region is labeled with a different integer. Parameter `mindim` defines the size of the smallest block allowed in the decomposition; this parameter must be a nonnegative integer power of 2, which allows decomposition down to regions of size 1×1 pixels, although this fine a detail normally is not used in practice.

Function `predicate` is a user-defined function. Its syntax is

```
flag = predicate(region)
```

This function must be written so that it returns `true` (a logical 1) if the pixels in `region` satisfy the predicate defined by the code in the function; otherwise, the value of `flag` must be `false` (a logical 0). Example 11.14 illustrates how to use this function.

Function `splitmerge` has a simple structure. First, the image is partitioned using function `qtdecomp`. Function `split_test` uses `predicate` to determine whether a region should be split. Because when a region is split into four it is not known which (if any) of the resulting four regions will pass the predicate test individually, it is necessary to examine the regions after the fact to see which regions in the partitioned image pass the test. Function `predicate` is used for this purpose also. Any quadregion that passes the test is filled with 1s. Any that does not is filled with 0s. A marker array is created by selecting one

element of each region that is filled with 1s. This array is used in conjunction with the partitioned image to determine region connectivity (adjacency); function `imreconstruct` is used for this purpose.

Function `splitmerge` follows. If necessary, the program pads the size of the input image to a square whose dimensions are the minimum integer power of 2 that encompasses the image. This allows function `qtdecomp` to split regions all the way down to size 1×1 (single pixels), as mentioned earlier.

splitmerge

```
function g = splitmerge(f, mindim, fun)
%SPLITMERGE Segment an image using a split-and-merge algorithm.
%   G = SPLITMERGE(F, MINDIM, @PREDICATE) segments image F by using
%   a split-and-merge approach based on quadtree decomposition.
%   MINDIM (a nonnegative integer power of 2) specifies the minimum
%   dimension of the quadtree regions (subimages) allowed. If
%   necessary, the program pads the input image with zeros to the
%   nearest square size that is an integer power of 2. This
%   guarantees that the algorithm used in the quadtree decomposition
%   will be able to split the image down to blocks of size 1-by-1.
%   The result is cropped back to the original size of the input
%   image. In the output, G, each connected region is labeled with a
%   different integer.
%
%   Note that in the function call we use @PREDICATE for the value
%   of fun.  PREDICATE is a a user-defined function. Its syntax is
%
%       FLAG = PREDICATE(REGION) Must return TRUE if the pixels in
%       REGION satisfy the predicate defined in the body of the
%       function; otherwise, the value of FLAG must be FALSE.
%
%   The following simple example of function PREDICATE is used in
%   Example 11.14 of the book.  It sets FLAG to TRUE if the
%   intensities of the pixels in REGION have a standard deviation
%   that exceeds 10, and their mean intensity is between 0 and 125.
%   Otherwise FLAG is set to false.
%
%       function flag = predicate(region)
%       sd = std2(region);
%       m = mean2(region);
%       flag = (sd > 10) & (m > 0) & (m < 125);

% Pad the image with zeros to the nearest square size that is an
% integer power of 2. This allows decomposition down to regions of
% size 1-by-1.
Q = 2^nextpow2(max(size(f)));
[M, N] = size(f);
f = padarray(f, [Q - M, Q - N], 'post');

% Perform splitting first.
Z = qtdecomp(f, @split_test, mindim, fun);
```

```
% Then, perform merging by looking at each quadregion and setting
% all its elements to 1 if the block satisfies the predicate defined
% in function PREDICATE.

% First, get the size of the largest block. Use full because Z is
% sparse.
Lmax = full(max(Z(:)));
% Next, set the output image initially to all zeros.  The MARKER
% array is used later to establish connectivity.
g = zeros(size(f));
MARKER = zeros(size(f));
% Begin the merging stage.
for K = 1:Lmax
   [vals, r, c] = qtgetblk(f, Z, K);
   if ~isempty(vals)
      % Check the predicate for each of the regions of size K-by-K
      % with coordinates given by vectors r and c.
      for I = 1:length(r)
         xlow = r(I); ylow = c(I);
         xhigh = xlow + K - 1; yhigh = ylow + K - 1;
         region = f(xlow:xhigh, ylow:yhigh);
         flag = fun(region);
         if flag
            g(xlow:xhigh, ylow:yhigh) = 1;
            MARKER(xlow, ylow) = 1;
         end
      end
   end
end

% Finally, obtain each connected region and label it with a
% different integer value using function bwlabel.
g = bwlabel(imreconstruct(MARKER, g));

% Crop and exit.
g = g(1:M, 1:N);

%-----------------------------------------------------------------------%
function v = split_test(B, mindim, fun)
% THIS FUNCTION IS PART OF FUNCTION SPLIT-MERGE. IT DETERMINES
% WHETHER QUADREGIONS ARE SPLIT. The function returns in v
% logical 1s (TRUE) for the blocks that should be split and
% logical 0s (FALSE) for those that should not.

% Quadregion B, passed by qtdecomp, is the current decomposition of
% the image into k blocks of size m-by-m.
```

```
% k is the number of regions in B at this point in the procedure.
k = size(B, 3);

% Perform the split test on each block. If the predicate function
% (fun) returns TRUE, the region is split, so we set the appropriate
% element of v to TRUE. Else, the appropriate element of v is set to
% FALSE.
v(1:k) = false;
for I = 1:k
   quadregion = B(:, :, I);
   if size(quadregion, 1) <= mindim
      v(I) = false;
      continue
   end
   flag = fun(quadregion);
   if flag
      v(I) = true;
   end
end
```



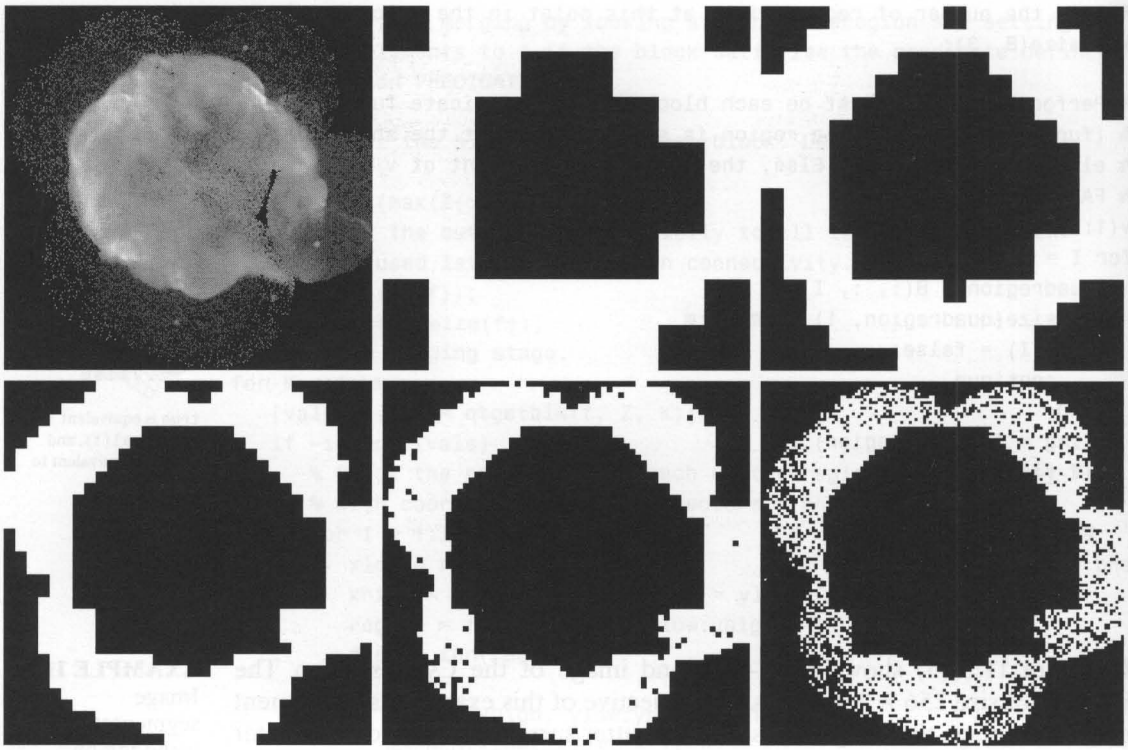
`true` is equivalent to `logical(1)`, and `false` is equivalent to `logical(0)`.

EXAMPLE 11.14: Image segmentation using region splitting and merging.

■ Figure 11.23(a) shows an X-ray band image of the Cygnus Loop. The image is of size 256 × 256 pixels. The objective of this example is to segment out of the image the "ring" of less dense matter surrounding the dense center. The region of interest has some obvious characteristics that should help in its segmentation. First, we note that the data has a random nature to it, indicating that its standard deviation should be greater than the standard deviation of the background (which is 0 because the background is constant) and of the large central region. Similarly, the mean value (average intensity) of a region containing data from the outer ring should be greater than the mean of the background (which is 0) and less than the mean of the large, lighter central region. Thus, we should be able to segment the region of interest by using these two parameters. In fact, the predicate function shown as an example in the documentation of function `splitmerge` contains this knowledge about the problem. The parameters shown in function `predicate` were determined by computing the mean and standard deviation of various subregions in Fig. 11.23(a).

Figures 11.23(b) through (f) show the results of segmenting Fig. 11.23(a) using function `splitmerge` with `mindim` values of 32, 16, 8, 4, and 2, respectively. All images show segmentation results with levels of boundary detail that are inversely proportional to the value of `mindim`.

All results in Fig. 11.23 are reasonable segmentations. If one of these images were to be used as a logical mask to extract the region of interest out of the original image, then the result in Fig. 11.23(d) would be the best choice because it is the solid region with the most detail. An important aspect of the method just illustrated is its ability to "capture" in function `predicate` information about a problem domain that can help in segmentation. ■



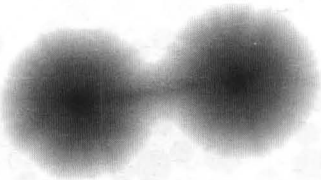
a b c
d e f

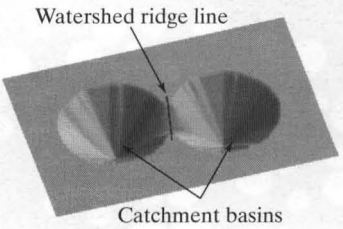
FIGURE 11.23 Image segmentation using a split-and-merge algorithm. (a) Original image. (b) through (f) Results of segmentation using function `splitmerge` with values of `mindim` equal to 32, 16, 8, 4, and 2, respectively. (Original image courtesy of NASA.)

11.5 Segmentation Using the Watershed Transform

In geography, a *watershed* is the ridge that divides areas drained by different river systems. A *catchment basin* is the geographical area draining into a river or reservoir. The *watershed transform* applies these ideas to gray-scale image processing in a way that can be used to solve a variety of image segmentation problems.

Understanding the watershed transform requires that we think of a gray-scale image as a topological surface, where the values of $f(x, y)$ are interpreted as heights. For example, we can visualize the simple image in Fig. 11.24(a) as the three-dimensional surface in Fig. 11.24(b). If we imagine rain falling on this surface, it is clear that water would collect in the two areas labeled as catchment basins. Rain falling exactly on the watershed ridge line would be equally likely to collect in either of the two catchment basins. The watershed transform finds the catchment basins and ridge lines in a gray-scale image. In terms of solving image segmentation problems, the key concept is to change the starting





a b

FIGURE 11.24
(a) Gray-scale scale image. (b) Image viewed as a surface, showing a watershed ridge line and catchment basins.

image into another image whose catchment basins are the objects or regions we want to identify.

Methods for computing the watershed transform are discussed in detail in Gonzalez and Woods [2008] and in Soille [2003]. The algorithm used in the Image Processing Toolbox is adapted from Meyer [1994].

11.5.1 Watershed Segmentation Using the Distance Transform

A tool used commonly in conjunction with the watershed transform for segmentation is the *distance transform*. The distance transform of a binary image is a relatively simple concept: It is the distance from every pixel to the nearest nonzero-valued pixel. For example, Fig. 11.25(a) shows a small binary image matrix, and Fig. 11.25(b) shows the corresponding distance transform. Note that every 1-valued pixel has a distance transform value of 0 because its closest nonzero pixel is itself. The distance transform can be computed using toolbox function `bwdist`, whose calling syntax is

```
D = bwdist(f)
```



EXAMPLE 11.15: Segmenting a binary image using the distance and watershed transforms.

■ In this example we show how the distance transform can be used with the toolbox watershed transform to segment circular blobs, some of which are touching each other. Specifically, we want to segment the processed dowel image, `f`, in Fig. 10.29(b). First, we convert the image to binary using `im2bw` and `graythresh`, as described in Section 11.3.1.

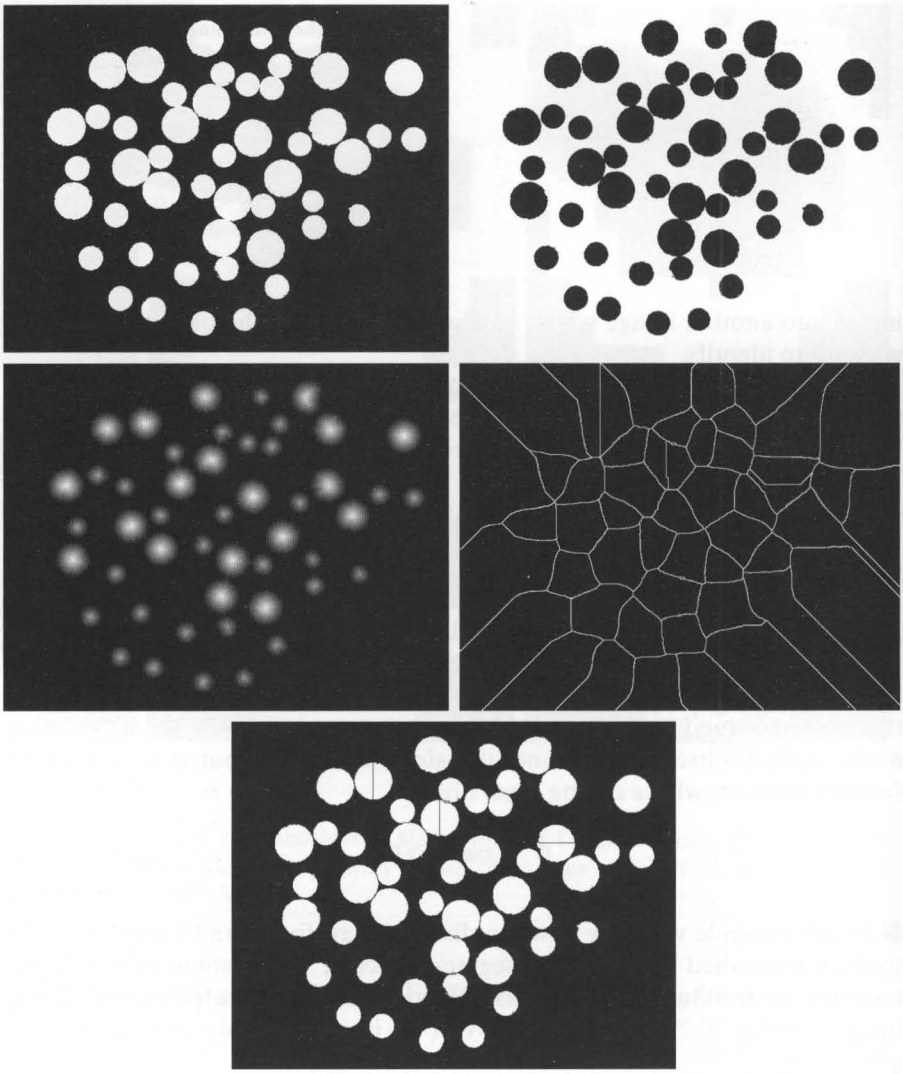
```
>> g = im2bw(f, graythresh(f));
```

Figure 11.26(a) shows the result. The next steps are to complement the image, compute its distance transform, and then compute the watershed transform of

1	1	0	0	0	0.00	0.00	1.00	2.00	3.00
1	1	0	0	0	0.00	0.00	1.00	2.00	3.00
0	0	0	0	0	1.00	1.00	1.41	2.00	2.24
0	0	0	0	0	1.41	1.00	1.00	1.00	1.41
0	1	1	1	0	1.00	0.00	0.00	0.00	1.00

a b

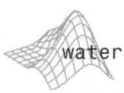
FIGURE 11.25
(a) Binary image. (b) Distance transform.



a b
c d
e

FIGURE 11.26
(a) Binary image. (b) Complement of image in (a). (c) Distance transform. (d) Watershed ridge lines of the negative of the distance transform. (e) Watershed ridge lines superimposed in black over original binary image. Some oversegmentation is evident.

the negative of the distance transform, using function `watershed`. The calling syntax for this function is




```
L = watershed(A, conn)
```

where `L` is a label matrix, as defined and discussed in Section 10.4, `A` is an input array (of any dimension in general, but two-dimensional in this chapter), and `conn` specifies connectivity [4 or 8 (the default) for 2-D arrays]. Positive integers in `L` correspond to catchment basins, and zero values indicate watershed ridge pixels:

```
>> gc = ~g;
>> D = bwdist(gc);
>> L = watershed(-D);
>> w = L == 0;
```

Figures 11.26(b) and (c) show the complemented image and its distance transform. Because 0-valued pixels of `L` are watershed ridge pixels, the last line of the preceding code computes a binary image, `w`, that shows only these pixels. This watershed ridge image is shown in Fig. 11.26(d). Finally, a logical AND of the original binary image and the complement of `w` serves to complete the segmentation, as shown in Fig. 11.26(e):

```
>> g2 = g & ~w;
```

Note that some objects in Fig. 11.20(e) were split improperly. This is called *oversegmentation* and is a common problem with watershed-based segmentation methods. The next two sections discuss different techniques for overcoming this difficulty. ■

11.5.2 Watershed Segmentation Using Gradients

The gradient magnitude is used often to preprocess a gray-scale image prior to using the watershed transform for segmentation. The gradient magnitude image has high pixel values along object edges, and low pixel values everywhere else. Ideally, then, the watershed transform would result in watershed ridge lines along object edges. The next example illustrates this concept.

EXAMPLE 11.16: Segmenting a gray-scale image using gradients and the watershed transform.

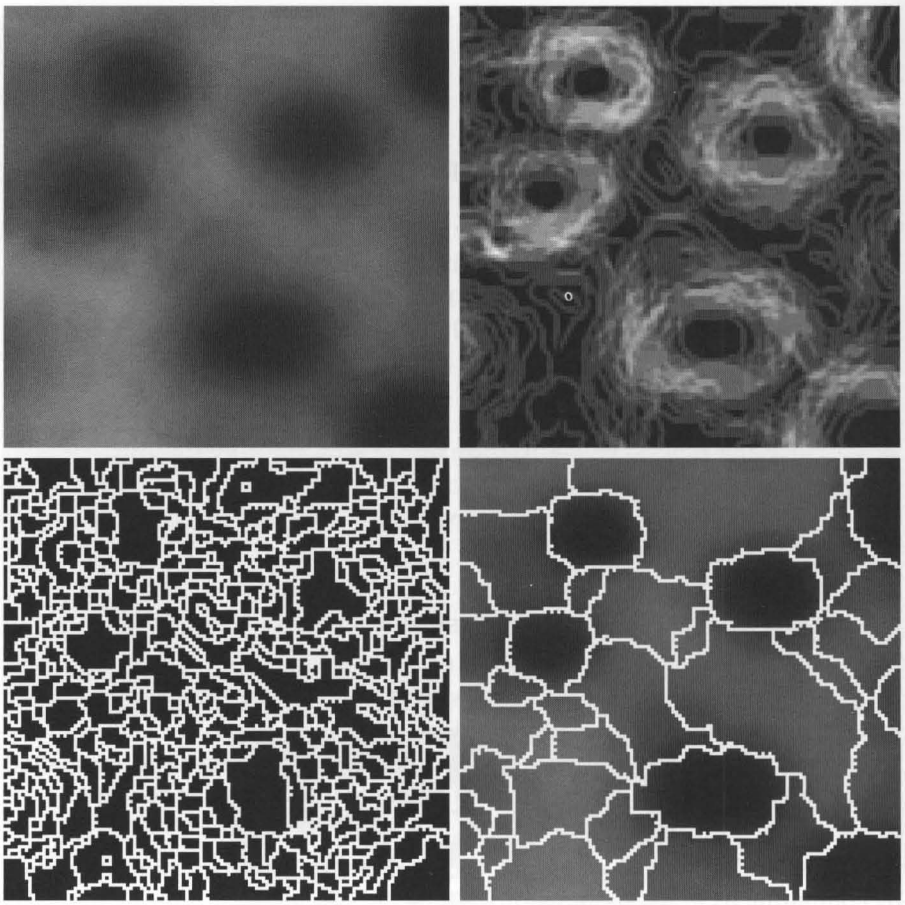
■ Figure 11.27(a) shows an image, `f`, containing several dark blobs. We start by computing its gradient magnitude, using either the linear filtering methods described in Section 11.1, or using a morphological gradient as described in Section 10.6.1.

```
>> h = fspecial('sobel');
>> fd = tofloat(f);
>> g = sqrt(imfilter(fd, h, 'replicate') .^ 2 + ...
           imfilter(fd, h', 'replicate') .^ 2);
```

Figure 11.27(b) shows the gradient magnitude image, `g`. Next we compute the watershed transform of the gradient and find the watershed ridge lines:

```
>> L = watershed(g);
>> wr = L == 0;
```

As Fig. 11.27(c) shows, this is not a good segmentation result; there are too many watershed ridge lines that do not correspond to the object boundaries of interest. This is another example of oversegmentation. One approach to this problem is to smooth the gradient image before computing its watershed



a b
c d

FIGURE 11.27
(a) Gray-scale image of small blobs. (b) Gradient magnitude image. (c) Watershed transform of (b), showing severe oversegmentation. (d) Watershed transform of the smoothed gradient image; some oversegmentation is still evident. (Original image courtesy of Dr. S. Beucher, CMM/ Ecole de Mines de Paris.)

transform. Here we use a close-opening, as described in Chapter 10:

```
>> g2 = imclose(imopen(g, ones(3,3)), ones(3,3));
>> L2 = watershed(g2);
>> wr2 = L2 == 0;
>> f2 = f;
>> f2(wr2) = 255;
```

The last two lines in the preceding code superimpose the watershed ridge lines in `wr` as white lines in the original image. Figure 11.27(d) shows the superimposed result. Although improvement over Fig. 11.27(c) was achieved, there are still some extraneous ridge lines, and it can be difficult to determine which catchment basins are actually associated with the objects of interest. The next section describes further refinements of watershed-based segmentation that deal with these difficulties. ■

11.5.3 Marker-Controlled Watershed Segmentation

As you saw in the previous section, direct application of the watershed transform to a gradient image can result in oversegmentation due to noise and other local irregularities of the gradient. The problems caused by these factors can be serious enough to render the result useless. In the present context, this means a large number of segmented regions. A practical solution to this problem is to limit the number of allowable regions by incorporating a preprocessing stage designed to bring additional knowledge into the segmentation process.

An approach used to control oversegmentation is based on the concept of markers. A *marker* is a connected component belonging to an image. We would like to have a set of *internal* markers that are inside each of the objects of interest, and a set of *external* markers that are contained in the background. These markers are used to modify the gradient image following the procedure described below in Example 11.17. Various methods have been suggested in the image processing literature for computing internal and external markers, many of which involve linear filtering, nonlinear filtering, and morphological processing, as described in previous chapters. Which method we choose for a particular application depends on the specific nature of the images associated with that application.

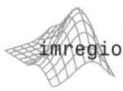
EXAMPLE 11.17: Illustration of marker-controlled watershed segmentation.

■ This example applies marker-controlled watershed segmentation to the electrophoresis gel image in Figure 11.28(a). We start by considering the results obtained from computing the watershed transform of the gradient image, without any other processing.

```
>> h = fspecial('sobel');
>> fd = tofloat(f);
>> g = sqrt(imfilter(fd, h, 'replicate') .^ 2 + ...
           imfilter(fd, h', 'replicate') .^ 2);
>> L = watershed(g);
>> wr = L == 0;
```

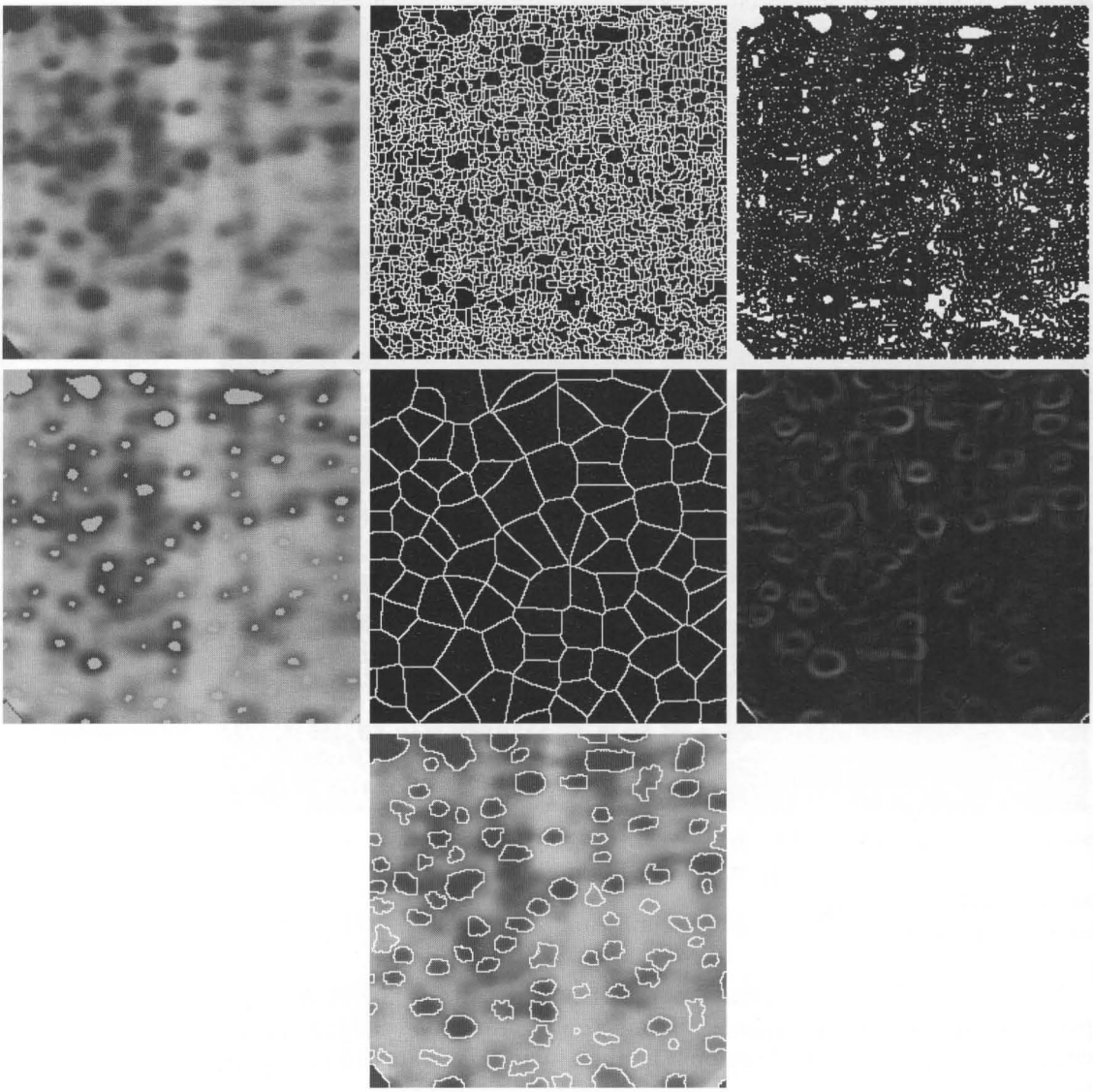
You can see in Fig. 11.28(b) that the result is severely oversegmented, due in part to the large number of regional minima. Toolbox function `imregionalmin` computes the location of all regional minima in an image. Its calling syntax is

```
rm = imregionalmin(f)
```



where `f` is a gray-scale image and `rm` is a binary image whose foreground pixels mark the locations of regional minima. We can use `imregionalmin` on the gradient image to see why the watershed function produces so many small catchment basins:

```
>> rm = imregionalmin(g);
```

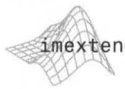


a b c
d e f
g

FIGURE 11.28 (a) Gel image. (b) Oversegmentation resulting from applying the watershed transform to the gradient magnitude image. (c) Regional minima of gradient magnitude. (d) Internal markers. (e) External markers. (f) Modified gradient magnitude. (g) Segmentation result. (Original image courtesy of Dr. S. Beucher, CMM/Ecole des Mines de Paris.)

Most of the regional minima locations shown in Fig. 11.28(c) are very shallow and represent detail that is irrelevant to our segmentation problem. To eliminate these extraneous minima we use toolbox function `imextendedmin`, which computes the set of "low spots" in the image that are deeper (by a certain height threshold) than their immediate surroundings. (See Soille [2003] for a detailed explanation of the *extended minima transform* and related operations.) The calling syntax for this function is

```
im = imextendedmin(f, h)
```



where `f` is a gray-scale image, `h` is the height threshold, and `im` is a binary image whose foreground pixels mark the locations of the deep regional minima. Here, we use function `imextendedmin` to obtain our set of internal markers:

```
>> im = imextendedmin(f, 2);
>> fim = f;
>> fim(im) = 175;
```

The last two lines superimpose the extended minima locations as gray blobs on the original image, as shown in Fig. 11.28(d). We see that the resulting blobs do a reasonably good job of "marking" the objects we want to segment.

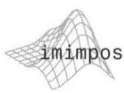
Next we must find external markers, or pixels that we are confident belong to the background. The approach we follow is to mark the background by finding pixels that are exactly midway between the internal markers. Surprisingly, we do this by solving another watershed problem; specifically, we compute the watershed transform of the distance transform of the internal marker image, `im`:

```
>> Lim = watershed(bwdist(im));
>> em = Lim == 0;
```

Figure 11.28(e) shows the resulting watershed ridge lines in the binary image `em`. Because these ridge lines are midway in between the dark blobs marked by `im`, they should serve well as our external markers.

We use both the internal and external markers to modify the gradient image using a procedure called *minima imposition*. The minima imposition technique (see Soille [2003] for details) modifies a gray-scale image so that regional minima occur only in marked locations. Other pixel values are "pushed up" as necessary to remove all other regional minima. Toolbox function `imimposemin` implements this technique. Its calling syntax is

```
mp = imimposemin(f, mask)
```



where `f` is a gray-scale image and `mask` is a binary image whose foreground pixels mark the desired locations of regional minima in the output image, `mp`. We modify the gradient image by imposing regional minima at the locations of both the internal and the external markers:

```
>> g2 = imimposemin(g, im | em);
```

Figure 11.28(f) shows the result. We are finally ready to compute the watershed transform of the marker-modified gradient image and look at the resulting watershed `ridgelines`:

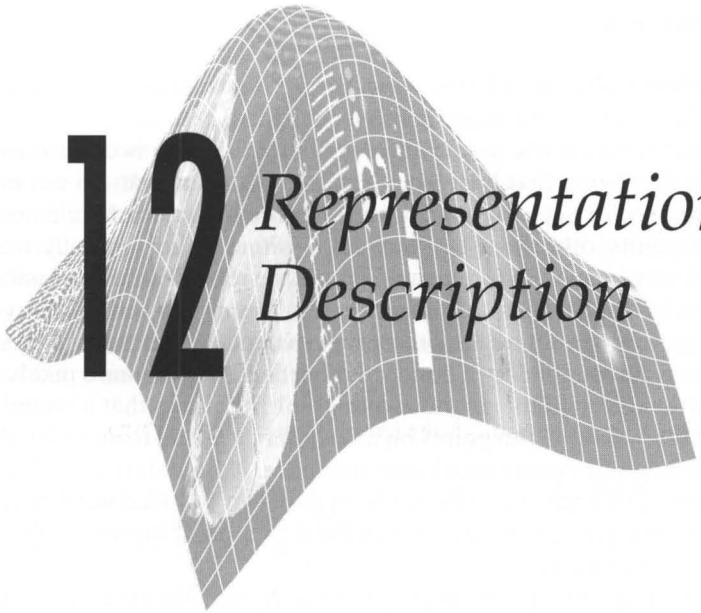
```
>> L2 = watershed(g2);
>> f2 = f;
>> f2(L2 == 0) = 255;
```

The last two lines superimpose the watershed ridge lines on the original image. The result, a much-improved segmentation, is shown in Fig. 11.28(g). ■

Marker selection can range from the simple procedures just described to considerably more complex methods involving size, shape, location, relative distances, texture content, and so on (see Chapter 12 regarding descriptors). The point is that using markers brings a priori knowledge to bear on the segmentation problem. Humans often aid segmentation and higher-level tasks in everyday vision by using a priori knowledge, one of the most familiar being the use of context. Thus, the fact that segmentation by watersheds offers a framework that can make effective use of this type of knowledge is a significant advantage of this method.

Summary

Image segmentation is an essential preliminary step in most automatic pictorial pattern recognition and scene analysis problems. As indicated by the range of methods and examples presented in this chapter, the choice of one segmentation technique over another is dictated mostly by the particular characteristics of the problem being considered. The methods discussed in this chapter, although far from being exhaustive, are representative of techniques used commonly in practice.



12 Representation and Description

Preview

After an image has been segmented into regions by methods such as those discussed in Chapter 11, the next step usually is to represent and describe the aggregate of segmented, "raw" pixels in a form suitable for further computer processing. Representing a region involves two basic choices: (1) We can represent the region in terms of its external characteristics (its boundary), or (2) we can represent it in terms of its internal characteristics (the pixels comprising the region). Choosing a representation scheme, however, is only part of the task of making the data useful to a computer. The next task is to *describe* the region based on the chosen representation. For example, a region may be *represented* by its boundary, and the boundary may be *described* by features such as its length and the number of concavities it contains.

An external representation is selected when interest is on shape characteristics. An internal representation is selected when the principal focus is on regional properties, such as color and texture. Both types of representations are used frequently in the same application. In either case, the features selected as descriptors should be as insensitive as possible to variations in size, translation, and rotation. Normalization for variations in intensity often is necessary as well. For the most part, the descriptors discussed in this chapter satisfy one or more of these properties.

12.1 Background

With reference to the discussion in Section 10.4, let S represent a subset of pixels in an image. Two pixels p and q are said to be *connected* in S if there exists a path between them consisting entirely of pixels in S. For any pixel p in S, the set of pixels connected to it in S is called a *connected component*. If it only has

one connected component, S is called a *connected set*. A subset, R, of pixels in an image is called a *region* of the image if R is a connected set.

In image processing applications connected components typically only have one component, so use of the term *connected component* generally refers to a region.

The *boundary* (also called the *border* or *contour*) of a region is defined as the set of pixels in the region that have one or more neighbors that are not in the region. As discussed in Section 10.1.2, points on a boundary or region are called *foreground* points; otherwise, they are *background* points. Initially we are interested in binary images, so foreground points are represented by 1s and background points by 0s. Later in this chapter we allow pixels to have gray-scale or multispectral values. Using the preceding concepts we define a *hole* as a background region surrounded by a connected border of foreground pixels.

See the discussion following function `bsubsamp` in Section 12.1.3 for a procedure to order a set of unordered boundary points.

From the definition given in the previous paragraph, it follows that a boundary is a connected set of points. The points on a boundary are said to be *ordered* if they form a clockwise or counterclockwise sequence. A boundary is said to be *minimally connected* if each of its points has exactly two 1-valued neighbors that are not 4-adjacent. An *interior* point is defined as a point anywhere in a region, except on its boundary.

Some of the functions in this chapter accept as inputs binary or numeric arrays. Recall from the discussion in Section 2.6.2 that a binary image in MATLAB refers *specifically* to a logical array of 0s and 1s. A numeric array can have any of the numeric classes defined in Table 2.3 (`uint8`, `double`, etc.). Recall also that a numeric array, `f`, is converted to logical using the function `logical(f)`. This function sets to 0 (false) all values in `f` that are 0, and to 1 (true) *all other values* in `f`. Toolbox functions that are designed to work only with binary images perform this conversion automatically on any non-binary input. Rather than introducing cumbersome notation to try to differentiate between functions that work only with binary inputs, it is preferable to let context be the guide as to the types of inputs accepted by a particular function. When in doubt, consult the help page for that function. Generally, we are specific as to the class of the result.

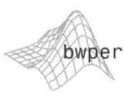
12.1.1 Functions for Extracting Regions and Their Boundaries

As discussed in Section 10.4, toolbox function `bwlabel` computes all the connected components (regions) in a binary image. We repeat its syntax here for convenience:

```
[L, num] = bwlabel(f, conn)
```

where `f` is the input image, `conn` specifies the desired connectivity (4 or 8, the latter being the default), `num` is the number of connected components found, and `L` is a label matrix that assigns to each connected component a unique integer from `1` to `num`. Recall from the discussion of Fig. 10.19 that the value of connectivity used can affect the number of regions detected.

Function `bwperim` with syntax




```
g = bwperim(f, conn)
```

returns a binary image, g, containing only the perimeter (boundary) pixels of all the regions in f. Unlike most functions in the Image Processing Toolbox, parameter conn in this particular function specifies the connectivity of the *background*: 4 (the default) or 8. Thus, to obtain 4-connected region boundaries we specify 8 for conn. Conversely, 8-connected boundaries result from specifying a value of 4 for conn. Function imfill, discussed in Section 12.1.2, has this characteristic also.

While bwperim produces a binary *image* containing the boundaries, function bwboundaries extracts the actual *coordinates* boundaries of all the regions in a binary image, f. Its syntax is

```
B = bwboundaries(f, conn, options)
```



See the bwboundaries help page for additional syntax forms.

where conn is with respect to the boundaries themselves, and can have the value 4 or 8 (the default). Parameter options can have the values 'holes' and 'noholes'. Using the first option extracts the boundaries of regions and holes. The boundaries of regions containing nested regions (referred to in the toolbox as *parent* and *child* regions) also are extracted. The second option results in only the boundaries of regions and their children. If only f and a value for conn are included in the argument, 'holes' is used as the default for options. If only f is included in the call, then 8 and 'holes' are used as defaults. The regions are listed first in B, followed by the holes (the third syntax below is used to find the number of regions and holes).

The output, B, is a $P \times 1$ cell array, where P is the number of objects (and holes, if so specified). Each cell in the cell array contains an $np \times 2$ matrix whose rows are the row and column coordinates of boundary pixels, and np is the number of boundary pixels for the corresponding region. The coordinates of each boundary are ordered in the *clockwise* direction, and the last point in a boundary is the same as the first, thus providing a closed boundary. Keeping in mind that B is a cell array, we change the order of travel of a boundary B{k} from clockwise to counterclockwise (and vice versa) using function flipud:

See Section 2.10.7 for a discussion of cell arrays.

```
Breversed{k} = flipud(B{k})
```

See Section 5.11.6 for an explanation of function flipud.

Another useful syntax for function bwboundaries is

```
[B, L] = bwboundaries(...)
```

In this case, L is a label matrix (of the same size as f) that labels each element of f (whether it is a region or a hole) with a different integer. Background pixels are labeled 0. The number of regions and holes is given by max(L(:)).

Finally, the syntax

```
[B, L, NR, A] = bwboundaries(...)
```

returns the number of regions found (NR) and a logical, sparse matrix A that details the parent-child-hole dependencies; that is, the most immediate boundary *enclosed* by B{k} is given by

See Section 2.8.7 regarding sparse matrices.

Function `find` is explained in Section 5.2.2.

```
boundaryEnclosed = find(A(:, k))
```

and, similarly, the most immediate boundary *enclosing* `B{k}` is given by

```
boundaryEnclosing = find(A(k, :))
```

(matrix `A` is explained in more detail in Example 12.1). The first `NR` entries in `B` are regions and the remaining entries (if any) are holes. The number of holes is given by `numel(B) - NR`.

It is useful to be able to construct and/or display a binary image that contains boundaries of interest. Given a boundary `b` in the form of an $np \times 2$ array of coordinates, where, as before, np is the number of points, the following custom function (see Appendix C for the listing):

`bound2im`

```
g = bound2im(b, M, N)
```

generates a binary image, `g`, of size $M \times N$, with 1s at the coordinates in `b` and a background of 0s. Typically, `M = size(f, 1)` and `N = size(f, 2)`, where `f` is the image from which `b` was obtained. In this way, `g` and `f` are registered spatially. If `M` and `N` are omitted, then `g` is the smallest binary image that encompasses the boundary while maintaining its original coordinate values.

If function `bwboundaries` finds multiple boundaries, we can get all the coordinates for use in function `bound2im` into a single array, `b`, of coordinates by concatenating the components of cell array `B`:

See Section 7.1.1 for an explanation of the `cat` operator. See also Example 12.13.

```
b = cat(1, B{:})
```

where the 1 indicates concatenation along the first (vertical) dimension. The following example illustrates the use of `bound2im` as an aid in visualizing the results of function `bwboundaries`.

EXAMPLE12.1: Using functions `bwboundaries` and `bound2im`.

■ Image `f` in Fig. 12.1(a) contains a region, a hole, and a single child, with the latter also containing a hole. The command

```
>> B = bwboundaries(f,'noholes');
```

extracts only the boundaries of regions using the default 8-connectivity. The command

```
>> numel(B)
ans
    2
```

indicates that two boundaries were found. Figure 12.1(b) shows a binary image containing these boundaries; the image was obtained using the commands:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 0   0 1 1 1 1 1 1 1 1 1 1 1 1 1 0   0 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 0   0 1 0 0 0 0 0 0 0 0 0 0 0 1 0   0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
0 1 1 0 0 0 0 0 0 0 0 1 1 1 0   0 1 0 0 0 0 0 0 0 0 0 0 0 1 0   0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
0 1 1 0 1 1 1 1 1 1 0 1 1 1 0   0 1 0 0 1 1 1 1 1 1 0 0 1 0 0   0 1 0 0 2 2 2 2 2 2 0 0 1 0
0 1 1 0 1 1 1 1 1 1 0 1 1 1 0   0 1 0 0 1 0 0 0 0 1 0 0 1 0 0   0 1 0 0 2 0 0 0 0 2 0 0 1 0
0 1 1 0 1 1 0 0 1 1 0 1 1 1 0   0 1 0 0 1 0 0 0 0 1 0 0 1 0 0   0 1 0 0 2 0 4 4 0 2 0 0 1 0
0 1 1 0 1 1 0 0 1 1 0 1 1 1 0   0 1 0 0 1 0 0 0 0 1 0 0 1 0 0   0 1 0 0 2 0 4 4 0 2 0 0 1 0
0 1 1 0 1 1 1 1 1 1 0 1 1 1 0   0 1 0 0 1 0 0 0 0 1 0 0 1 0 0   0 1 0 0 2 0 0 0 0 2 0 0 1 0
0 1 1 0 1 1 1 1 1 1 0 1 1 1 0   0 1 0 0 1 1 1 1 1 1 0 0 1 0 0   0 1 0 0 2 2 2 2 2 2 0 0 1 0
0 1 1 0 0 0 0 0 0 0 0 1 1 1 0   0 1 0 0 0 0 0 0 0 0 0 0 1 0 0   0 1 0 0 0 0 0 0 0 0 0 0 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 0   0 1 0 0 0 0 0 0 0 0 0 0 1 0 0   0 1 0 0 0 0 0 0 0 0 0 0 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 0   0 1 1 1 1 1 1 1 1 1 1 1 1 0 0   0 1 1 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

a b c

FIGURE 12.1 (a) Original array containing two regions (1-valued pixels) and two holes. (b) Boundaries of regions, extracted using function `bwboundaries` and displayed as an image using function `bound2im`. (c) Boundaries of regions and of the innermost hole.

```
>> b = cat(1, B{:});
>> [M, N] = size(f);
>> image = bound2im(b, M, N)
```

The command

```
>> [B, L, NR, A] = bwboundaries(f);
```

extracts the boundaries of all regions and holes using the default 8-connectivity. The total number of region and hole boundaries extracted is given by

```
>> numel(B)
ans =
     4
```

and the number of holes is

```
>> numel(B) - NR
ans =
     2
```

We can use function `bound2im` in combination with `L` to display the boundaries of regions and/or holes. For example,

```
>> bR = cat(1, B{1:2}, B{4});
>> imageBoundaries = bound2im(bR, M, N);
```

is a binary image containing 1s in the boundary of regions and the boundary of the last hole. Then, the command

```
>> imageNumberedBoundaries = imageBoundaries.*L
```

displays the numbered boundaries, as Fig. 12.1(c) shows. If, instead, we had wanted to display *all* the numbered boundaries, we would have used the command

```
>> bR = cat(1, B{:});
>> imageBoundaries = bound2im(bR, M, N);
>> imageNumberedBoundaries = imageBoundaries.*L
```

For larger images, visualization is aided by color-coding the boundaries (the `bwboundaries` help page shows several examples of this).

Finally, we take a brief look at matrix `A`. The number of boundaries enclosed, for example, by `B{1}` is

```
>> find(A(:, 1))
ans =
     3
```

and the number of boundaries enclosing `B{1}` is

```
>> find(A(1, :))
ans =
   Empty matrix: 1-by-0
```

as expected, because `B{1}` is the outermost boundary. The elements of `A` are

```
>> A
A =
   (3,1)        1
   (4,2)        1
   (2,3)        1
```

In the language of sparse matrices, this says that elements (3, 1), (4, 2), and (2, 3) are 1; all other elements are zero. We can see this by looking at the full matrix:

```
>> full(A)
ans =
     0     0     0     0
     0     0     1     0
     1     0     0     0
     0     1     0     0
```

Reading down column k, a 1 in row n indicates that the most immediate boundary *enclosed* by B{k} is boundary number n. Reading across row k, a 1 in column m indicates that the most immediate boundary *enclosing* B{k} is boundary m. Note that this notation does not differentiate between boundaries of regions and holes. For example, boundary 2 (second column of A) encloses boundary 4 (fourth row of A), which we know is the boundary of the innermost hole. ■

12.1.2 Some Additional MATLAB and Toolbox Functions Used in This Chapter

Function `imfill` was mentioned briefly in Section 10.5.2. This function performs differently for binary and intensity image inputs so, to help clarify the notation in this section, we let `fB` and `fI` represent binary and intensity images, respectively. If the output is a binary image, we denote it by `gB`; otherwise we denote it as `g`. The syntax

```
gB = imfill(fB, locations, conn)
```

performs a flood-fill operation on background pixels (i.e., it changes background pixels to 1) of the input binary image `fB`, starting from the points specified in `locations`. This parameter can be an $nL \times 1$ vector (nL is the number of locations), in which case it contains the *linear indices* (see Section 2.8.2) of the starting coordinate locations. Parameter `locations` can be an $nL \times 2$ matrix also, in which case each row contains the 2-D coordinates of one of the starting locations in `fB`. As is the case with function `bwperim`, parameter `conn` specifies the connectivity to be used on the *background* pixels: 4 (the default), or 8. If both `locations` and `conn` are omitted from the input argument, the command

```
gB = imfill(fB)
```

displays the binary image, `fB`, on the screen and lets the user select the starting locations using the mouse. Click the left mouse button to add points. Press **BackSpace** or **Delete** to remove the previously selected point. A shift-click, right-click, or double-click selects a final point and then starts the fill operation. Pressing **Return** finishes the selection without adding a point.

Using the syntax

```
gB = imfill(fB, conn, 'holes')
```

fills holes in the input binary image. Parameter `conn` is as above.

The syntax

```
g = imfill(fI, conn)
```

fills holes in an input intensity image, `fI`. In this syntax, a *hole* is an area of dark pixels surrounded by lighter pixels, and parameter `'holes'` is not used.

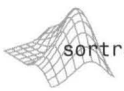
Function `find` can be used in conjunction with `bwlabel` to return vectors of coordinates for the pixels that make up a specific object. For example, if

```
[gB, num] = bwlabel(fB)
```

yields more than one connected region (i.e., `num` > 1), we obtain the coordinates of, say, the second region using

```
[r c] = find(gB == 2)
```

As indicated earlier, the 2-D coordinates of regions or boundaries are organized in this chapter in the form of $np \times 2$ arrays, where each row is an (x, y) coordinate pair, and np is the number of points in the region or boundary. In some cases it is necessary to sort these arrays. Function `sortrows` can be used for this purpose:




```
z = sortrows(S)
```

This function sorts the rows of `S` in ascending order. Argument `S` must be either a matrix or a column vector. In this chapter, `sortrows` is used only with $np \times 2$ arrays. If several rows have identical first coordinates, they are sorted in ascending order of the second coordinate. If we want to sort the rows of `S` and also eliminate duplicate rows, we use function `unique`, which has the syntax




```
[z, m, n] = unique(S, 'rows')
```

where `z` is the sorted array with no duplicate rows, and `m` and `n` are such that `z = S(m, :)` and `S = z(n, :)`. For example, if `S = [1 2; 6 5; 1 2; 4 3]`, then `z = [1 2; 4 3; 6 5]`, `m = [3; 4; 2]`, and `n = [1; 3; 1; 2]`. Note that `z` is arranged in ascending order and that `m` indicates which rows of the original array were kept.

If it is necessary to shift the rows of an array up, down, or sideways, use function `circshift`:




```
z = circshift(S, [ud lr])
```

where `ud` is the number of elements by which `S` is shifted up or down. If `ud` is positive, the shift is down; otherwise it is up. Similarly, if `lr` is positive, the array is shifted to the right `lr` elements; otherwise it is shifted to the left. If only up and down shifting is needed, we can use a simpler syntax

```
z = circshift(S, ud)
```

If `S` is an image, `circshift` is nothing more than the familiar *scrolling* (up and down) or *panning* (right and left), with the image wrapping around.

12.1.3 Some Basic Utility M-Functions

Tasks such as converting between regions and boundaries, ordering boundary points in a contiguous chain of coordinates, and subsampling a boundary to

simplify its representation and description are typical of the processes that we employ routinely in this chapter. The following custom utility M-functions are used for these purposes. To avoid a loss of focus on the main topic of this chapter, we discuss only the syntax of these functions. The documented code for each custom function is included in Appendix C. As noted earlier, boundaries are represented as $np \times 2$ arrays in which each row represents a 2-D pair of coordinates.

Function `bound2eight` with syntax

```
b8 = bound2eight(b)
```

`bound2eight`

removes from boundary `b` the pixels that are necessary for 4-connectedness, leaving a boundary with pixels are only 8-connected. It is required that `b` be a closed, connected set of pixels ordered sequentially in the clockwise or counterclockwise direction. The same conditions apply to function `bound2four`:

```
b4 = bound2four(b)
```

`bound2four`

This function inserts new boundary pixels wherever there is a diagonal connection, thus producing an output boundary in which pixels are 4-connected.

Function

```
[s, su] = bsubsamp(b, gridsep)
```

`bsubsamp`

subsamples a (single) boundary `b` onto a grid whose lines are separated by `gridsep` pixels. The output `s` is a boundary with fewer points than `b`, the number of such points being determined by the value of `gridsep`. Output `su` is the set of boundary points scaled so that transitions in their coordinates are unity. This is useful for coding the boundary using chain codes, as discussed in Section 12.2.1. It is required in the preceding three functions that the points in `b` be ordered in a clockwise or counterclockwise direction (the outputs are in the same order as the input). If the points in `b` are not ordered sequentially (but they are points of a fully-connected boundary), we can convert `b` to a clockwise sequence using the commands:

```
>> image = bound2im(b);
>> b = bwboundaries(image, 'noholes');
```

That is, we convert the boundary to a binary image and then use function `bwboundaries` to extract the boundary as a clockwise sequence. If a counterclockwise sequence is desired, we let `b = flipud(b)`, as mentioned earlier.

When a boundary is subsampled using `bsubsamp`, its points cease to be connected. They can be reconnected by using

```
z = connectpoly(s(:, 1), s(:, 2))
```

`connectpoly`

where `s(:, 1)` and `s(:, 2)` are the horizontal and vertical coordinates of the subsampled boundary, respectively. It is required that the points in `s` be ordered,

either in a clockwise or counterclockwise direction. The rows of output z are the coordinates of a connected boundary formed by connecting the points in s with straight line segments (see function intline below). The coordinates in output z are in the same direction as the coordinates in s.

Function connectpoly is useful for producing a polygonal, fully connected boundary that generally is simpler to describe than the original boundary, b, from which s was obtained. Function connectpoly is useful also when working with functions that generate only the vertices of a polygon, such as function im2minperpoly, discussed in Section 12.2.3.

Computing the integer coordinates of a straight line joining two points is a basic tool when working with boundaries. Toolbox function intline is well suited for this purpose. Its syntax is




```
[x y] = intline(x1, x2, y1, y2)
```

intline is an undocumented Image Processing Toolbox utility function. Its code is included in Appendix C.

where (x1, y1) and (x2, y2) are the integer coordinates of the two points to be connected. The outputs x and y are column vectors containing the integer x- and y-coordinates of the straight line joining the two points.

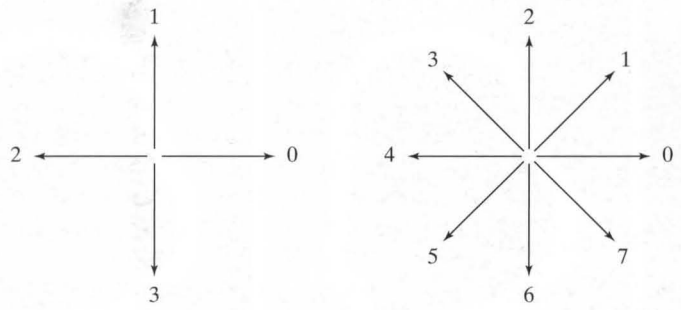
12.2 Representation

As noted at the beginning of this chapter, the segmentation techniques discussed in Chapter 11 generally yield raw data in the form of pixels along a boundary or pixels contained in a region. Although these data sometimes are used directly to obtain descriptors (as in determining the texture of a region), standard practice is to use schemes that compact the data into representations that are considerably more useful in the computation of descriptors. In this section we discuss the implementation of various representation approaches.

12.2.1 Chain Codes

Chain codes are used to represent a boundary by a connected sequence of straight-line segments of specified length and direction. Typically, this representation is based on 4- or 8-connectivity of the segments. The direction of each segment is coded by using a numbering scheme such as the ones in Figs. 12.2(a) and (b). Chain codes based on this scheme are referred to as *Freeman chain codes*.

The chain code of a boundary depends on the starting point. However, the code can be normalized with respect to the starting point by treating it as a circular sequence of direction numbers and redefining the starting point so that the resulting sequence of numbers forms an integer of *minimum magnitude*. We can normalize for rotation [in increments of 90° or 45° for the codes in Figs. 12.2(a) and (b)] by using the *difference* of the chain code instead of the code itself. The difference is obtained by counting the number of direction changes (in a counterclockwise direction in Fig. 12.2) that separate two adjacent elements of the code. For instance, the first difference of the 4-direction chain



a b

FIGURE 12.2 Direction numbers for (a) a 4-directional chain code, and (b) an 8-directional chain code.

code 10103322 is 3133030. Treating the code as a circular sequence, the first element of the difference is computed by using the transition between the last and first components of the chain; the result is 33133030 for the preceding code. Normalization with respect to arbitrary rotation angles is achieved by orienting the boundary with respect to some dominant feature, such as its major axis, as discussed in Section 12.3.2, or its principal-component vector, as discussed at the end of Section 12.5.

Function `fchcode` (see Appendix C), with syntax

```
c = fchcode(b, conn, dir)
```

`fchcode`

computes the Freeman chain code of an $np \times 2$ set of *ordered* boundary points stored in array `b`. Output `c` is a structure with the following fields, where the numbers inside the parentheses indicate array size:

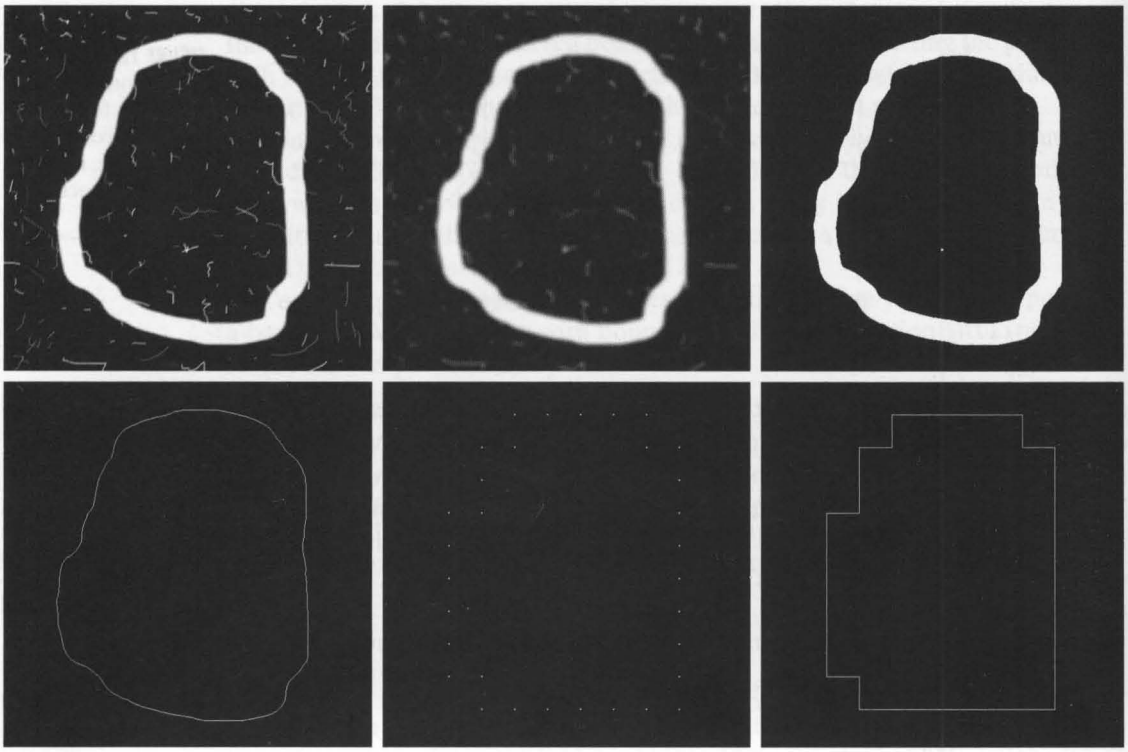
See Section 2.10.7 for a discussion of structures.

`c.fcc`	=	Freeman chain code ($1 \times np$)
`c.diff`	=	First difference code of `c.fcc` ($1 \times np$)
`c.mm`	=	Integer of minimum magnitude ($1 \times np$)
`c.diffmm`	=	First difference of code `c.mm` ($1 \times np$)
`c.x0y0`	=	Coordinates were the code starts (1×2)

Parameter `conn` specifies the connectivity of the code; its value can be 4 or 8 (the default). A value of 4 is valid only when the boundary contains no diagonal transitions. Parameter `dir` specifies the direction of the output code. If `'same'` is specified, the code is in the same direction as the points in `b`. Using `'reverse'` causes the code to be in the opposite direction. The default is `'same'`. Thus, writing `c = fchcode(b, conn)` uses the default direction, and `c = fchcode(b)` uses the default connectivity and direction.

EXAMPLE 12.2: Freeman chain code and some of its variations.

■ Figure 12.3(a) shows a 570×570 image, `f`, of a circular stroke embedded in specular noise. The objective of this example is to obtain the chain code and first difference of the object's outer boundary. It is evident by looking at Fig. 12.3(a) that the noise fragments attached to the object would result in a very irregular boundary, not truly descriptive of the general shape of the object.



a b c
d e f

FIGURE 12.3 (a) Noisy image. (b) Image smoothed with a 9×9 averaging mask. (c) Thresholded image. (d) Boundary of binary image. (e) Subsampled boundary. (f) Connected points from (e).

Smoothing generally is a routine process when working with noisy boundaries. Figure 12.3(b) shows the result, g, of using a 9×9 averaging mask:

```
>> h = fspecial('average', 9);
>> g = imfilter(f, h, 'replicate');
```

The binary image in Fig. 12.3(c) was then obtained by thresholding:

```
>> gB = im2bw(g, 0.5);
```

The (outer) boundaries of `gB` were computed using function `bwboundaries` discussed in the previous section:

```
>> B = bwboundaries(gB, 'noholes');
```

As in the illustration in Section 12.1.1, we are interested in the longest boundary (the inner dot in Fig. 12.3(c) also has a boundary):

```
>> d = cellfun('length', B);
>> [maxd, k] = max(d);
>> b = B{k};
```

See Section 2.10.2 for an explanation of this use of function max.

The boundary image in Fig. 12.3(d) was generated using the commands:

```
>> [M N] = size(g);
>> g = bound2im(b, M, N);
```

Obtaining the chain code of b directly would result in a long sequence with small variations that are not necessarily representative of the general shape of the boundary. Thus, as is typical in chain-code processing, we subsample the boundary using function bsubsamp discussed in the previous section:

```
>> [s, su] = bsubsamp(b, 50);
```

We used a grid separation equal to approximately 10% the width of the image. The resulting points can be displayed as an image [Fig. 12.3(e)]:

```
>> g2 = bound2im(s, M, N);
```

or as a connected sequence [Fig. 12.2(f)] by using the commands

```
>> cn = connectpoly(s(:, 1), s(:, 2));
>> g3 = bound2im(cn, M, N);
```

The advantage of using this representation, as opposed to Fig. 12.3(d), for chain-coding purposes is evident by comparing the two figures. The chain code is obtained from the scaled sequence su:

```
>> c = fchcode(su);
```

This command resulted in the following outputs:

```
>> c.x0y0
ans =
    7    3

>> c.fcc
ans =
2 2 0 2 2 0 2 0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2 2

>> c.mm
ans =
0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2 2 2 2 0 2 2 0 2
```

```
>> c.diff
ans =
0 6 2 0 6 2 6 0 0 0 6 2 6 0 0 0 0 0 0 6 0 0 0 0 0 6 2 6 0 0 0

>> c.diffmm
ans =
0 0 0 6 2 6 0 0 0 0 0 0 6 0 0 0 0 0 6 2 6 0 0 0 0 6 2 0 6 2 6
```

By examining `c.fcc`, Fig. 12.3(f), and `c.x0y0`, we see that the code starts on the left of the figure and proceeds in the clockwise direction, which is the same direction as the coordinates of the original boundary. ■

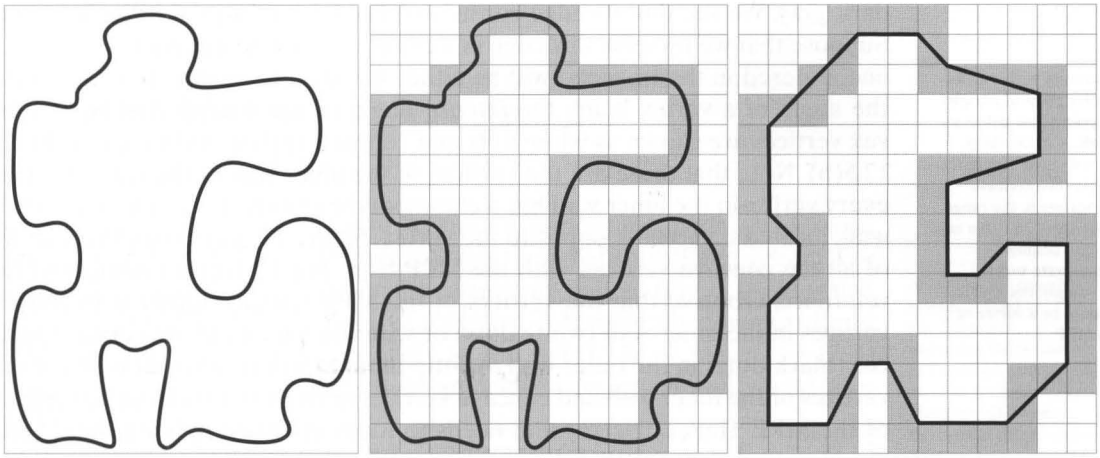
12.2.2 Polygonal Approximations Using Minimum-Perimeter Polygons

A digital boundary can be approximated with arbitrary accuracy by a polygon. For a closed boundary, the approximation becomes exact when the number of vertices of the polygon is equal to the number of points in the boundary, and each vertex coincides with a point on the boundary. The goal of a polygonal approximation is to capture the essence of the shape in a given boundary using the fewest possible number of vertices. This problem is not trivial in general and can quickly turn into a time-consuming iterative search. However, approximation techniques of modest complexity are well suited for image processing tasks. Among these, one of the most powerful is representing a boundary by a *minimum-perimeter polygon* (MPP), as defined in the following discussion.

Foundation

An intuitively appealing approach for generating an algorithm to compute MPPs is to enclose a boundary [Fig. 12.4(a)] by a set of concatenated cells, as in Fig. 12.4(b). Think of a boundary as a (continuous) rubber band. As it is allowed to shrink, the rubber band will be constrained by the inner and outer walls of the bounding region defined by the cells. Ultimately, this shrinking produces the shape of a polygon of minimum perimeter (with respect to this geometrical arrangement) that circumscribes the region enclosed by the cell strip, as Fig. 12.4(c) shows. Note in this figure that all the vertices of the MPP coincide with corners of either the inner or the outer wall of cells.

The size of the cells determines the accuracy of the polygonal approximation. In the limit, if the size of each (square) cell corresponds to a pixel in a digital representation of the boundary, the maximum error between each vertex of the MPP and the closest point in the original boundary would be $\sqrt{2}d$, where d is the minimum possible distance between pixels (i.e., the distance between pixels established by the resolution of the original sampling grid). This error can be reduced in half by forcing each cell in the polygonal approximation to be centered on its corresponding pixel in the sampled boundary. The objective is to use the largest possible cell size acceptable in a given application, thus

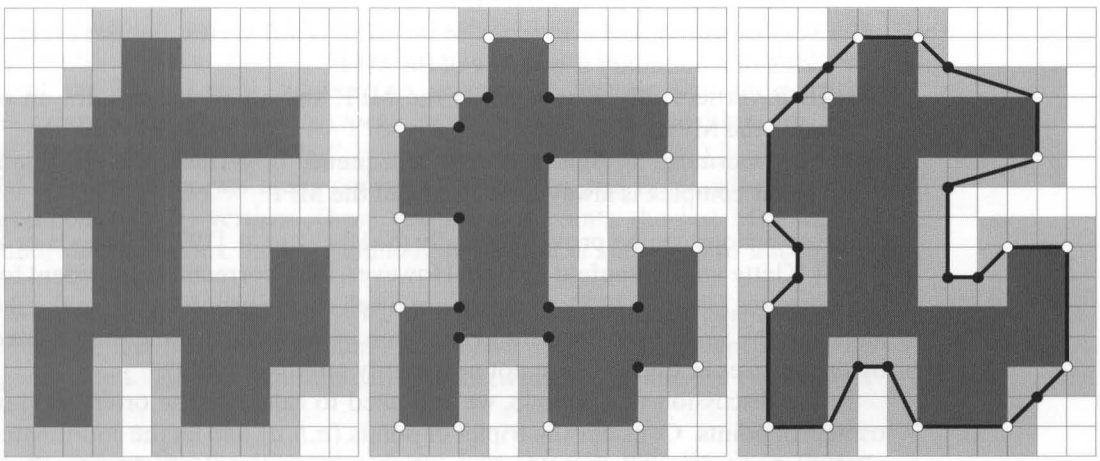


a b c

FIGURE 12.4 (a) An object boundary (black curve). (b) Boundary enclosed by cells (in gray). (c) Minimum-perimeter polygon obtained by allowing the boundary to shrink. The vertices of the polygon in (c) are created by the corners of the inner and outer walls of the gray region.

producing MPPs with the fewest number of vertices. Our goal in this section is to formulate and implement a procedure for finding these MPP vertices.

The cellular approach just described reduces the shape of the object enclosed by the original boundary to the shape of the region circumscribed by the inner wall of the cells in Fig. 12.4(b). Figure 12.5(a) shows this shape in



a b c

FIGURE 12.5 (a) Region (dark gray) resulting from enclosing the original boundary by cells (see Fig. 12.4). (b) Convex (white dots) and concave (black dots) vertices obtained by following the boundary of the dark gray region in the counterclockwise direction. (c) Concave vertices (black dots) displaced to their diagonal mirror locations in the outer wall of the bounding region. The MPP (black curve) is superimposed for reference.

dark gray. We see that its edge consists of 4-connected straight line segments. Suppose that we traverse this edge in a counterclockwise direction. Every turn encountered in the traversal will be either a *convex* or a *concave* vertex, with the angle of a vertex being the *interior* angle of the 4-connected edge. Convex vertices are shown as white dots and concave vertices as black dots in Fig. 12.5(b). Note that these are the vertices of the inner wall of the cells, and that every vertex in the inner wall has a corresponding "mirror" vertex in the outer wall, located diagonally opposite the vertex. Figure 12.5(c) shows the mirrors of all the concave vertices, with the MPP from Fig. 12.4(c) superimposed for reference. Observe that the vertices of the MPP coincide either with convex vertices in the inner wall (white dots) or with the mirrors of the concave vertices (black dots) in the outer wall. A little thought will reveal that only convex vertices of the inner wall and concave vertices of the outer wall can be vertices of the MPP. Thus, our algorithm needs to focus attention only on these vertices.

A convex vertex is the center point of a triplet of points that define an angle, θ, in the range $0° < \theta < 180°$; similarly, angles of a concave vertex are in the range $180° < \theta < 360°$. An angle of 180° defines a degenerate vertex (a straight line), which cannot be a vertex of an MPP.

An Algorithm for Finding MPPs

The set of cells enclosing a boundary is called a *cellular complex*. We assume that the boundaries under consideration are not self intersecting, a condition that leads to *simply connected* cellular complexes. Based on these assumptions, and letting *white* (W) and *black* (B) denote *convex* and *mirrored concave* vertices, respectively, we state the following observations:

1. The MPP bounded by a simply connected cellular complex is not self intersecting.
2. Every convex vertex of the MPP is a W vertex, but not every W vertex of a boundary is a vertex of the MPP.
3. Every mirrored concave vertex of the MPP is a B vertex, but not every B vertex of a boundary is a vertex of the MPP.
4. All B vertices are on or outside the MPP, and all W vertices are on or inside the MPP.
5. The uppermost, leftmost vertex in a sequence of vertices contained in a cellular complex is always a W vertex of the MPP.

These assertions can be proved formally (Sklansky et al. [1972]; Sloboda et al. [1998]; Klette and Rosenfeld [2004]). However, their correctness is evident for our purposes (see Fig. 12.5), so we do not dwell on the proofs here. Unlike the angles of the vertices of the dark gray region in Fig. 12.5, the angles sustained by the vertices of the MPP are not necessarily multiples of 90°.

In the discussion that follows, we will need to calculate the orientation of triplets of points. Consider the triplet of points, (a, b, c), and let the coordinates of these points be $a = (x_a, y_b)$, $b = (x_b, y_b)$, and $c = (x_c, y_c)$. If we arrange these points as the rows of the matrix

$$\mathbf{A} = \begin{bmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{bmatrix}$$

then it follows from matrix analysis that

$$\det(\mathbf{A}) = \begin{cases} > 0 & \text{if } (a,b,c) \text{ is a counterclockwise sequence} \\ = 0 & \text{if the points are collinear} \\ < 0 & \text{if } (a,b,c) \text{ is a clockwise sequence} \end{cases}$$

where $\det(\mathbf{A})$ is the determinant of $\mathbf{A}$, and movement in a counterclockwise or clockwise direction is with respect to a right-handed coordinate system. For example, using our right-handed image coordinate system (Fig. 2.1) (in which the origin is at the top left, the positive x-axis extends vertically downward, and the positive y-axis extends horizontally to the right), the sequence $a = (3,4)$, $b = (2,3)$, and $c = (3,2)$ is in the counterclockwise direction and would give $\det(\mathbf{A}) > 0$.

Assuming the coordinate system defined in Fig. 2.1, when traversing the boundary of a polygon in the counterclockwise direction, all points to the right of the direction of travel are *outside* the polygon. All points to the left of the direction of travel are *inside* the polygon.

It is convenient to define

$$\text{sgn}(a,b,c) \equiv \det(\mathbf{A})$$

so that $\text{sgn}(a,b,c) > 0$ for a counterclockwise sequence, $\text{sgn}(a,b,c) < 0$ for a clockwise sequence, and $\text{sgn}(a,b,c) = 0$ when the points are collinear. Geometrically, $\text{sgn}(a,b,c) > 0$ indicates that point c lies on the positive side of the line passing through points a and b; $\text{sgn}(a,b,c) < 0$ indicates that point c lies on the negative side of that line; and $\text{sgn}(a,b,c) = 0$ indicates that point c is on the line.

To prepare the data for the MPP algorithm we form a list whose rows are the coordinates of each vertex, and note whether a vertex is W or B. The concave vertices must be mirrored, as in Fig. 12.5(c); the vertices must be in sequential order; and the first vertex in the sequence must be the uppermost, leftmost vertex, which we know from property 5 is a W vertex of the MPP. Let V_0 denote this vertex. We assume that the vertices are arranged in the *counterclockwise* direction. The algorithm for finding MPPs uses two "crawler" points: a white crawler (W_C) and a black (B_C) crawler. W_C crawls along convex (W) vertices, and B_C crawls along mirrored concave (B) vertices. These two crawler points, the last MPP vertex found, and the vertex being examined are all that is necessary to implement the procedure.

See Section 12.1.3 for a procedure to order a list of unordered vertices.

We start by setting $W_C = B_C = V_0$. Then, at any step in the algorithm, let V_L denote the last MPP vertex found, and let V_k denote the current vertex being examined. Three conditions can exist between V_L, V_k, and the two crawler points:

(a) V_k lies to the positive side of the line through pair (V_L, W_C); that is, $\text{sgn}(V_L, W_C, V_k) > 0$.

(b) V_k lies on the negative side of the line though pair (V_L, W_C) or is collinear with it; that is $\text{sgn}(V_L, W_C, V_k) \leq 0$. At the same time, V_k lies to the positive side of the line through (V_L, B_C) or is collinear with it; that is, $\text{sgn}(V_L, B_C, V_k) \geq 0$.

(c) V_k lies on the negative side of the line though pair (V_L, B_C); that is, $\text{sgn}(V_L, B_C, V_k) < 0$.

If condition (a) holds, the next MPP vertex is W_C and we let $V_L = W_C$; then we reinitialize the algorithm by setting $W_C = B_C = V_L$, and continue with the next vertex after V_L.

If condition (b) holds, V_k becomes a *candidate* MPP vertex. In this case, we set $W_C = V_k$ if V_k is convex (i.e., it is a W vertex); otherwise we set $B_C = V_k$ and continue with the next vertex in the list.

If condition (c) holds, the next MPP vertex is B_C and we let $V_L = B_C$; then we reinitialize the algorithm by setting $W_C = B_C = V_L$, and continue with the next vertex after V_L.

The algorithm terminates when it reaches the first vertex again, and has thus processed all the vertices in the polygon. It has been proved (Sloboda et al. [1998]; Klette and Rosenfeld [2004]) that this algorithm finds all the MPP vertices of a polygon enclosed by a simply-connected cellular complex.

Some of the M-Functions Used to Implement the MPP Algorithm

We use function `qtdecomp` introduced in Section 11.4.2 as the first step in obtaining the cellular complex enclosing a boundary. As usual, we consider the region, `B`, in question to be composed of 1s and the background of 0s. We are interested in the following syntax:

```
Q = qtdecomp(B, threshold, [mindim maxdim])
```

See Section 2.8.7 regarding sparse matrices.

where `Q` is a sparse matrix containing the quadtree structure. If `Q(k, m)` is nonzero, then `(k, m)` is the upper-left corner of a block in the decomposition and the size of the block is `Q(k, m)`.

A block is split if the maximum value of the block elements minus the minimum value of the block elements is greater than `threshold`. The value of this parameter is specified between 0 and 1, independently of the class of the input image. Using the preceding syntax, function `qtdecomp` will not produce blocks smaller than `mindim` or larger than `maxdim`. Blocks larger than `maxdim` are split even if they do not meet the threshold condition. The ratio `maxdim/mindim` must be a power of 2. If only one of the two values is specified (without the brackets), the function assumes that it is `mindim`. This is the formulation we use in this section.

Image `B` must be of size `K` × `K`, such that the ratio of `K/mindim` is an integer power of 2. It follows that the smallest possible value of `K` is the largest dimension of `B`. The size requirements generally are met by padding `B` with zeros with option `'post'` in function `padarray`. For example, suppose that `B` is of size 640 × 480 pixels, and we specify `mindim = 3`. Parameter `K` has to satisfy the conditions `K >= max(size(B))` and `K/mindim = 2^p`, or `K = mindim*(2^p)`. Solving for `p` gives `p = 8`, in which case `K = 768`.

To obtain the block values in a quadtree decomposition we use function `qtgetblk`, discussed in Section 10.4.2:

```
[vals, r, c] = qtgetblk(B, Q, mindim)
```

where `vals` is an array containing the values of the `mindim` × `mindim` blocks in the quadtree decomposition of `B`, and `Q` is the sparse matrix returned by `qtdecomp`. Parameters `r` and `c` are vectors containing the row and column coordinates of the upper-left corners of the blocks.

EXAMPLE 12.3: Obtaining the cellular complex enclosing the boundary of a region.

■ With reference to the image in Fig. 12.6(a), suppose that we specify `mindim = 2`. The image is of size 32 × 32 and it is easily verified that no additional padding is required for the specified value of `mindim`. The 4-connected boundary of the region was obtained using the following command:

```
>> g = bwperim(f, 8);
```

Figure 12.6(b) shows the result. Note that `g` is still an image, which now contains only a 4-connected boundary.

Recall from the discussion in Section 12.1.1 that to obtain 4-connected boundaries we specify 8-connectivity for the background.

Figure 12.6(c) shows the quadtree decomposition of `g`, resulting from the command

```
>> Q = qtdecomp(g, 0, 2);
```

where 0 was used for the threshold so that blocks were split down to the minimum 2 × 2 size specified, regardless of the mixture of 1s and 0s they contained (each such block is capable of containing between zero and four pixels). Note that there are numerous blocks of size greater than 2 × 2, but they are all homogeneous.

Next we used `qtgetblk(g, Q, 2)` to extract the values and top-left corner coordinates of all the blocks of size 2 × 2. Then, all the blocks that contained at least one pixel valued 1 were filled with 1s using `qtsetblk`. This result, which we denote by `gF`, is shown in Fig. 12.6(d). The dark cells in this image constitute the cellular complex.

Figure 12.6(e) shows in gray the region bounded by the cellular complex. This region was obtained using the command

```
>> R = imfill(gF, 'holes') & g;
```

We are interested in the 4-connected boundary of this region, which we obtain using the commands

```
>> B = bwboundaries(R, 4, 'noholes');
>> b  = B{1}; % There is only one boundary in this case.
```

Figure 12.6(f) shows the result. The direction numbers in the figure are part of the Freeman chain code of the boundary, obtained using function `fchcode`. ■

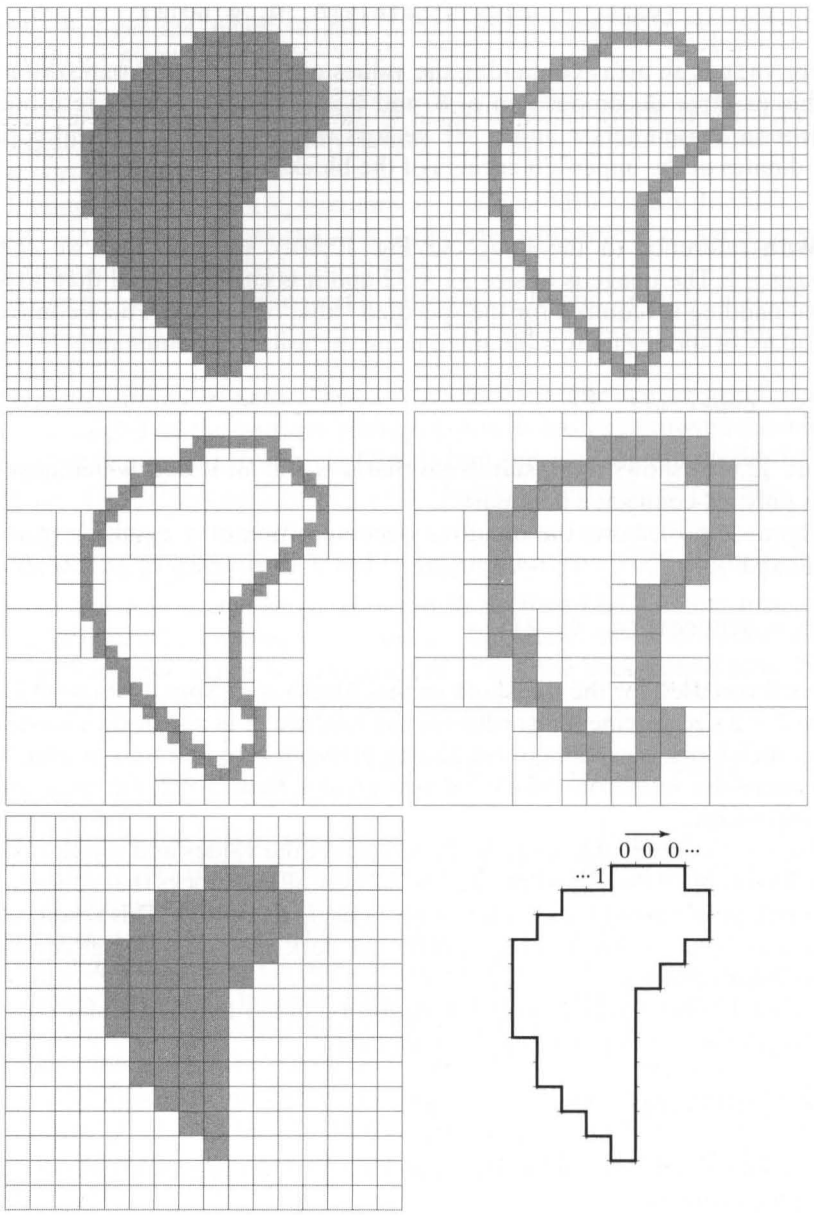
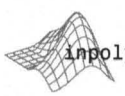


FIGURE 12.6
(a) Original image (the small squares denote individual pixels). (b) 4-connected boundary. (c) Quadtree decomposition using square blocks of size 2 pixels. (d) Result of filling with 1s all blocks of size 2×2 that contained at least one element valued 1. This is the cellular complex. (e) Inner region of (d). (f) 4-connected boundary points obtained using function `bwboundaries`. The numbers shown are part of the chain code.

Sometimes it is necessary to determine if a point lies inside or outside a polygonal boundary. Function `inpolygon` can be used for this purpose:




```
IN = inpolygon(X, Y, xv, yv)
```

where X and Y are vectors containing the x- and y-coordinates of the points to be tested, and xv and yv are vectors containing the x- and y-coordinates of the polygon vertices, arranged in a clockwise or counterclockwise sequence. Output IN is a vector whose length is equal to the number of points being tested. Its values are 1 for points inside or on the boundary of the polygon, and 0 for points outside the boundary.

An M-Function for Computing MPPs

The MPP algorithm is implemented by custom function im2minperpoly, whose listing is included in Appendix C. The syntax is

```
[X, Y, R] = im2minperpoly(f, cellsize)
```

im2minperpoly

where f is an input binary image containing a *single* region or boundary, and cellsize specifies the size of the square cells in the cellular complex used to enclose the boundary. Column vectors X and Y contain the x- and y-coordinates of the MPP vertices. Output R is a binary image of the region enclosed by the cellular complex [e.g, see Fig. 12.6(e)].

EXAMPLE 12.4: Using function im2minperpoly.

■ Figure 12.7(a) is a binary image, f, of a maple leaf, and Fig. 12.7(b) shows the boundary obtained using the commands

```
>> B = bwboundaries(f, 4, 'noholes');
>> b = B{1};
>> [M, N] = size(f);
>> bOriginal = bound2im(b, M, N);
>> imshow(bOriginal)
```

This is the reference boundary against which various MMPs are compared in this example. Figure 12.7(c) is the result of using the commands

```
>> [X, Y] = im2minperpoly(f, 2);
>> b2 = connectpoly(X, Y);
>> bCellsize2 = bound2im(b2, M, N);
>> figure, imshow(bCellsize2)
```

Similarly, Figs. 12.7(d) through (f) show the MPPs obtained using square cells of sizes 3, 4, and 8. The thin stem is lost with cells larger than 2×2 as a result of lower resolution. The second major shape characteristic of the leaf is its set of three main lobes. These are preserved reasonably well even for cells of size 8, as Fig. 12.7(f) shows. Further increases in the size of the cells to 10 and even to 16 still preserve this feature, as Figs. 12.8(a) and (b) show. However, as Figs. 12.8(c) and (d) demonstrate, values of 20 and higher cause this characteristic to be lost. ■

a b
c d
e f

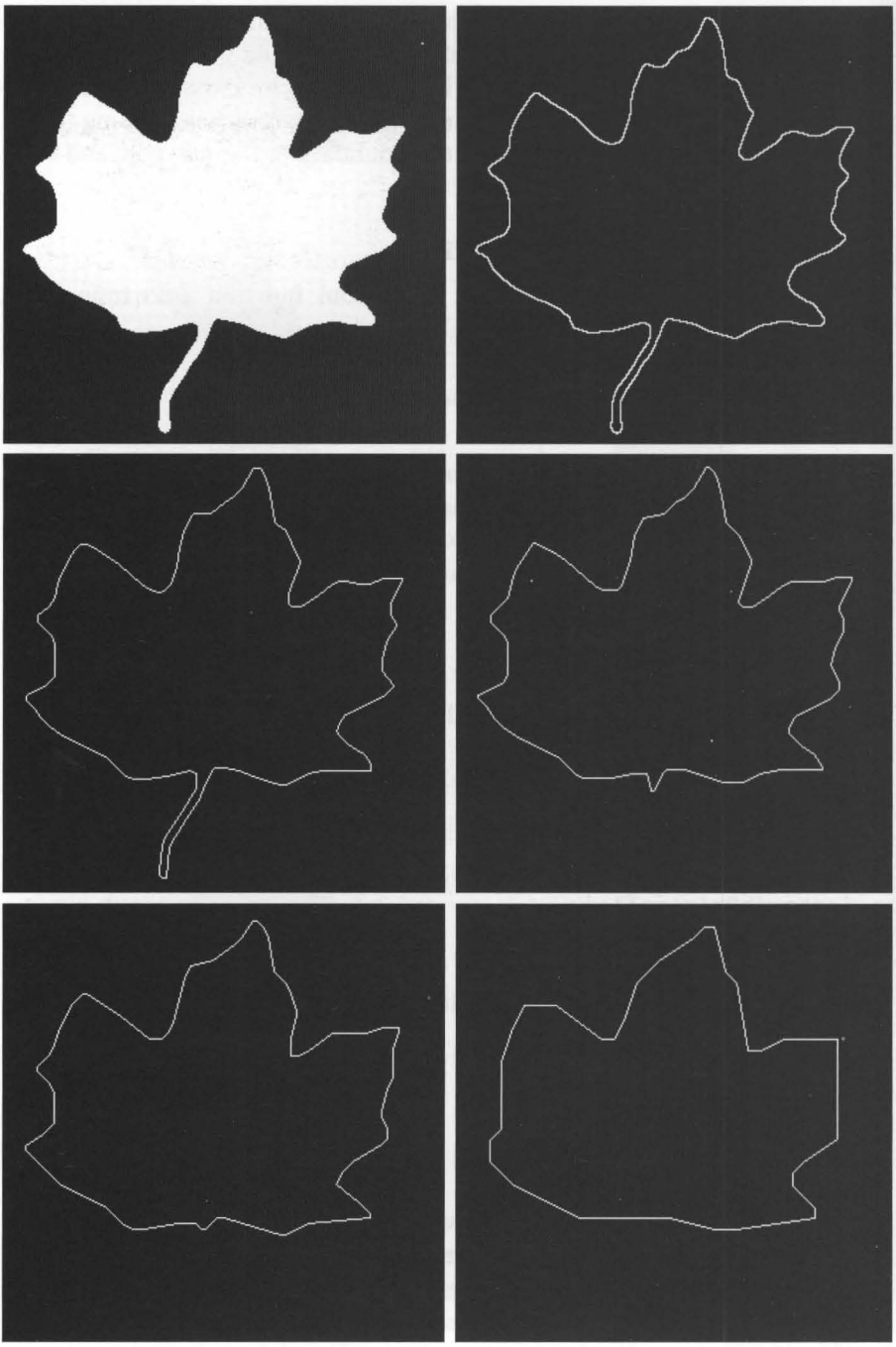
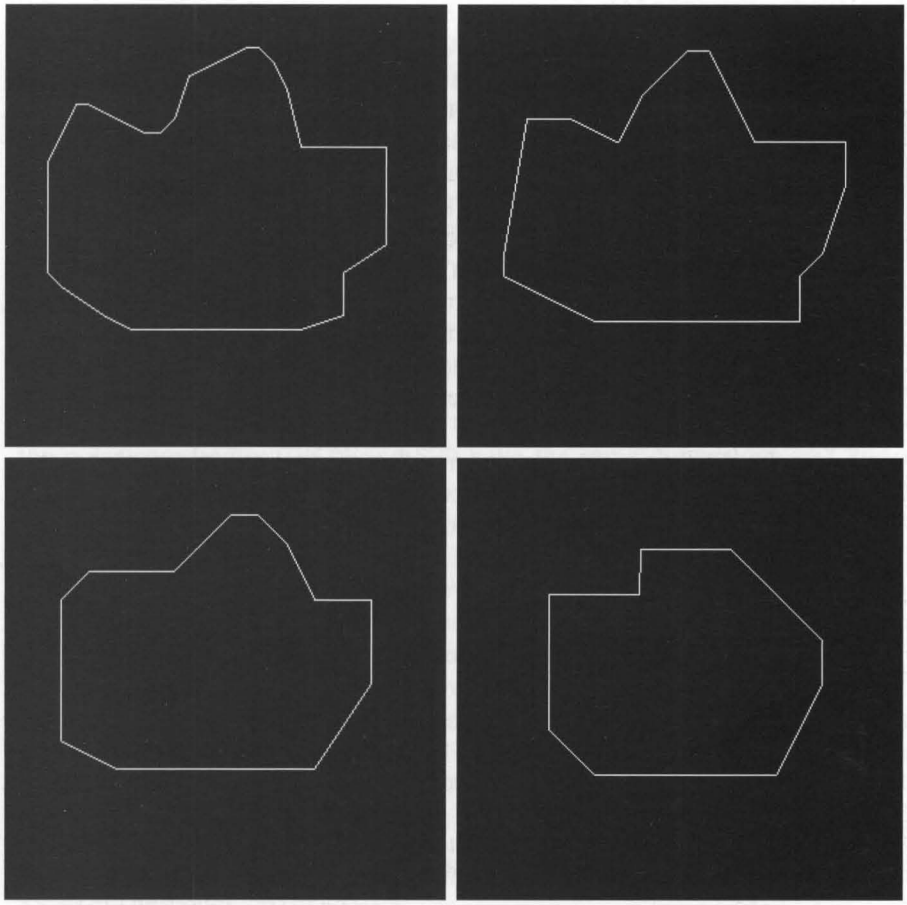


FIGURE 12.7
(a) Original image of size 312×312 pixels. (b) 4-connected boundary. (c) MPP obtained using square bounding cells of size 2. (d) through (f) MPPs obtained using square cells of sizes 3, 4, and 8, respectively.



a b
c d

FIGURE 12.8
MPPs obtained with even larger bounding square cells of sizes (a) 10, (b) 16, (c) 20, and (d) 32.

12.2.3 Signatures

A *signature* is a 1-D functional representation of a boundary and may be generated in various ways. One of the simplest is to plot the distance from an interior point (e.g., the centroid) to the boundary as a function of angle, as in Fig. 12.9. Regardless of how a signature is generated, however, the basic idea is to reduce the boundary representation to a 1-D function, which presumably is easier to describe than the original 2-D boundary. It makes sense to use signatures only when it can be guaranteed that the vector extending from its origin to the boundary intersects the boundary only once, thus yielding a single-valued function of increasing angle. This excludes boundaries with self-intersections, and (typically) boundaries with deep, narrow concavities or thin, long protrusions.

Signatures generated by the approach just described are invariant to translation, but they do depend on rotation and scaling. Normalization with respect to rotation can be achieved by finding a way to select the same starting point to generate the signature, regardless of the shape's orientation. One way to

do so is to select the starting point as the point farthest from the origin of the vector (see Section 12.3.1), if this point happens to be unique and reasonably independent of rotational aberrations for each shape of interest.

Another way is to select a point on the major eigen axis (see Example 12.15). This method requires more computation but is more rugged because the direction of the eigen axes is obtained using all contour points. Yet another way is to obtain the chain code of the boundary and then use the approach discussed in Section 12.1.2, assuming that the rotation can be approximated by the discrete angles in the code directions defined in Fig. 12.1.

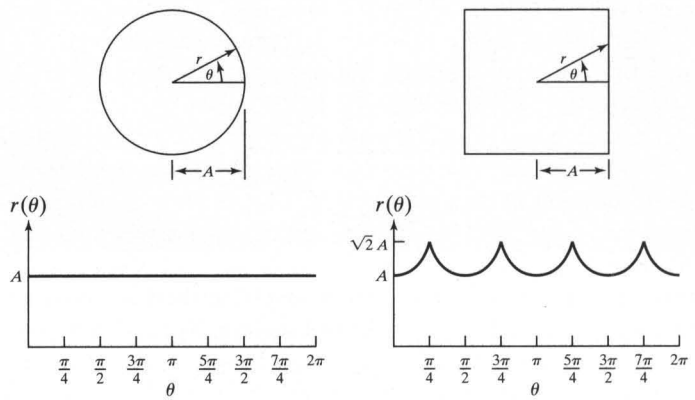
Based on the assumptions of uniformity in scaling with respect to both axes, and that sampling is taken at equal intervals of θ, changes in size of a shape result in changes in the amplitude values of the corresponding signature. One way to normalize for this dependence is to scale all functions so that they always span the same range of values, say, [0, 1]. The main advantage of this method is simplicity, but it has the potentially serious disadvantage that scaling of the entire function is based on only two values: the minimum and maximum. If the shapes are noisy, this can be a source of error from object to object. A more rugged approach is to divide each sample by the variance of the signature, assuming that the variance is not zero—as is the case in Fig. 12.9(a)—or so small that it creates computational difficulties. Use of the variance yields a variable scaling factor that is inversely proportional to changes in size and works much as automatic gain control does. Whatever the method used, keep in mind that the basic idea is to remove dependency on size while preserving the fundamental shape of the waveforms.

Function `signature` (see Appendix C), finds the signature of a boundary. Its syntax is

`signature`

```
[dist, angle] = signature(b, x0, y0)
```

where `b` is an $np \times 2$ array whose rows contain the x and y coordinates of the boundary points, ordered in a clockwise or counterclockwise direction. In the input, `(x0, y0)` are the coordinates of the point from which the distance to the boundary is measured. If `x0` and `y0` are not included in the argument,



a b
c d

FIGURE 12.9
(a) and (b) Circular and square objects. (c) and (d) Corresponding distance-versus-angle signatures.

`signature` uses the coordinates of the centroid of the boundary by default. The amplitude of the signature [i.e., the distance from `(x0, y0)` to the boundary] as a function of increasing angle is output in `dist`. The maximum size of arrays `dist` and `angle` is 360×1 indicating a maximum resolution of one degree. The input to function `signature` must be a one-pixel-thick boundary obtained, for example, using function `bwboundaries` discussed earlier. As before, we assume that a boundary is a closed curve.

Function `signature` utilizes MATLAB's function `cart2pol` to convert Cartesian to polar coordinates. The syntax is

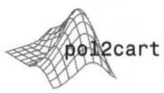
```
[THETA, RHO] = cart2pol(X, Y)
```



where `X` and `Y` are vectors containing the coordinates of the Cartesian points. The vectors `THETA` and `RHO` contain the corresponding angle and length of the polar coordinates. `THETA` and `RHO` have the same dimensions as `X` and `Y`. Figure 12.10 shows the convention used by MATLAB for coordinate conversions. Note that the MATLAB coordinates `(X, Y)` in this function are related to our image coordinates (x, y) as `X` $= y$ and `Y` $= -x$ [see Fig. 2.1(a)].

Function `pol2cart` is used for converting back to Cartesian coordinates:

```
[X, Y] = pol2cart(THETA, RHO)
```



EXAMPLE 12.5: Signatures.

■ Figures 12.11(a) and (b) show two images, `fsq` and `ftr`, containing an irregular square and a triangle, respectively. Figure 12.11(c) shows the signature of the square, obtained using the commands

```
>> bSq = bwboundaries(fsq, 'noholes');
>> [distSq, angleSq] = signature(bSq{1});
>> plot(angleSq, distSq)
```

A similar set of commands yielded the plot in Fig. 12.11(d). Simply counting the number of prominent peaks in the two signatures is sufficient to differentiate between the fundamental shape of the two boundaries. ■

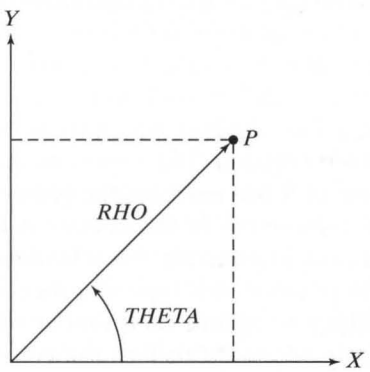
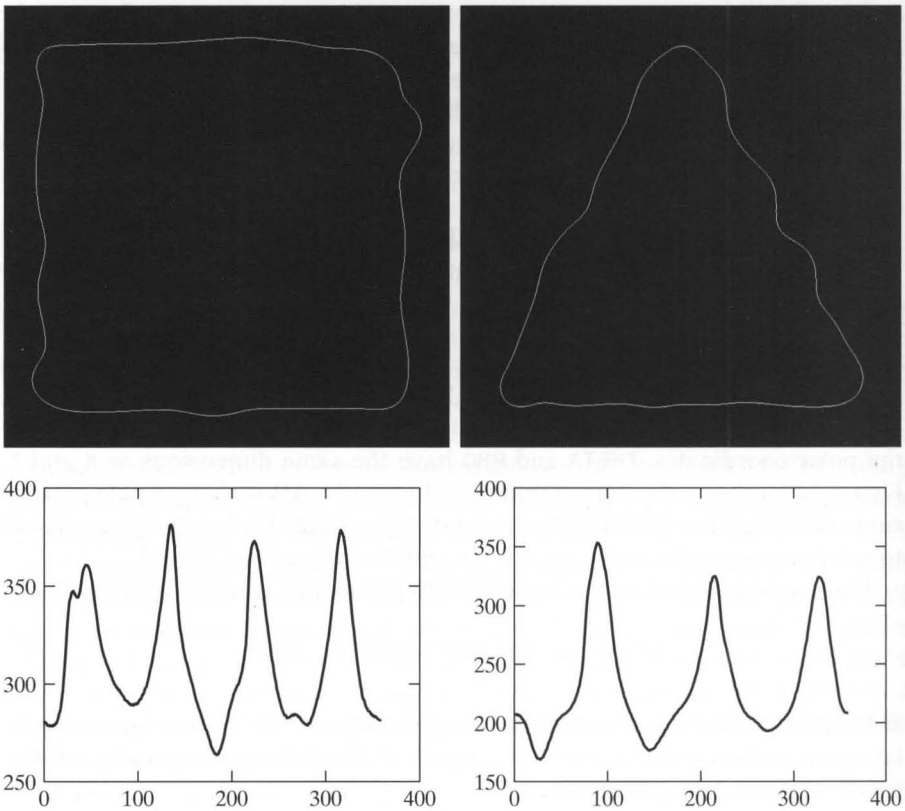


FIGURE 12.10 Axis convention used by MATLAB for performing conversions between polar and Cartesian coordinates, and vice versa.



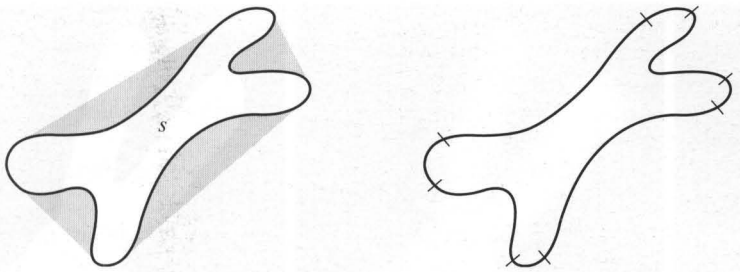
a b
c d

FIGURE 12.11
(a) and (b) Boundaries of an irregular square and triangle. (c) and (d) Corresponding signatures.

12.2.4 Boundary Segments

Decomposing a boundary into segments reduces the boundary's complexity and generally simplifies the description process. This approach is attractive when the boundary contains one or more significant concavities that carry shape information. In this case, using the convex hull of the region enclosed by the boundary is a powerful tool for robust decomposition of the boundary.

The *convex hull*, H, of an arbitrary set S is the smallest convex set containing S. The set difference, $H - S$, is called the *convex deficiency*, D, of the set S. To see how these concepts can be used to partition a boundary into meaningful segments, consider Fig. 12.12(a), which shows an object (set S) and its convex deficiency (shaded regions). The region boundary can be partitioned by following the contour of S and marking the points at which a transition is made into or out of a component of the convex deficiency. Figure 12.12(b) shows the result in this case. In principle, this scheme is independent of region size and orientation. In practice, this type of processing is preceded typically by aggressive smoothing to reduce the number of "insignificant" concavities. The MATLAB tools necessary to find the convex hull and implement



a b

FIGURE 12.12
(a) A region S and its convex deficiency (shaded). (b) Partitioned boundary.

boundary decomposition in the manner just described are contained in function `regionprops`, discussed in Section 12.4.1.

12.2.5 Skeletons

An important approach for representing the structural shape of a planar region is to reduce it to a graph. This reduction may be accomplished by obtaining the *skeleton* of the region via a thinning (also called *skeletonizing*) algorithm.

The skeleton of a region may be defined via the *medial axis transformation* (MAT). The MAT of a region R with border b is as follows. For each point p in R, we find its closest neighbor in b. If p has more than one such neighbor, it is said to belong to the *medial axis* (*skeleton*) of R.

Although the MAT of a region is an intuitive concept, direct implementation of this definition is expensive computationally, as it involves calculating the distance from every interior point to every point on the boundary of a region. Numerous algorithms have been proposed for improving computational efficiency while at the same time attempting to approximate the medial axis representation of a region.

As noted in Section 10.3.4, the Image Processing Toolbox generates an image containing the skeletons of all regions in a binary image `B` via function `bwmorph`, using the following syntax:

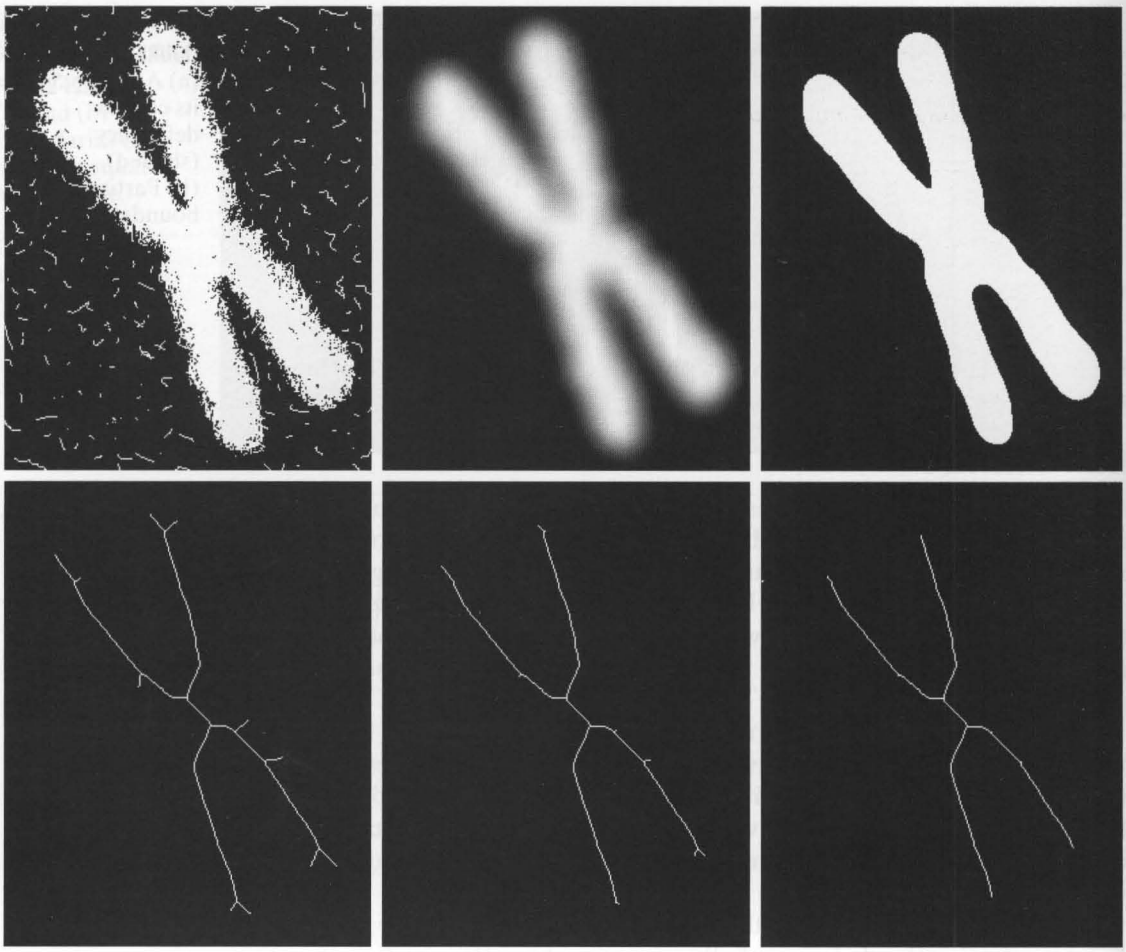
```
skeletonImage = bwmorph(B, 'skel', Inf)
```

This function removes pixels on the boundaries of objects but does not allow objects to break apart.

EXAMPLE 12.6: Computing the skeleton of a region.

■ Figure 12.13(a) shows a 344×270 image, `f`, representative of what a human chromosome looks like after it has been segmented out of an electron microscope image with magnification on the order of 30,000X. The objective of this example is to compute the skeleton of the chromosome.

Clearly, the first step in the process must be to isolate the chromosome from the background of irrelevant detail. One approach is to smooth the image and then threshold it. Figure 12.13(b) shows the result of smoothing `f` using a 25×25 Gaussian spatial mask with `sig = 15`:



a b c
d e f

FIGURE 12.13 (a) Segmented human chromosome. (b) Image smoothed using a 25×25 Gaussian averaging mask with `sig` = 15. (c) Thresholded image. (d) Skeleton. (e) Skeleton after eight applications of spur removal. (f) Result of seven additional applications of spur removal.

```
>> h = fspecial('gaussian', 25, 15);
>> g = imfilter(f, h, 'replicate');
>> imshow(g) % Fig. 12.13(b)
```

Next, we threshold the smoothed image:

```
>> g = im2bw(g, 1.5*graythresh(g));
>> figure, imshow(g) % Fig. 12.13(c)
```

where the automatically-determined threshold, `graythresh(g)`, was multiplied

by 1.5 to increase by 50% the amount of thresholding. The reasoning for this is that increasing the threshold value increases the amount of data removed from the boundary, thus further reducing noise. The skeleton of Fig. 12.13(d) was obtained using the command

```
>> s = bwmorph(g, 'skel', Inf); % Fig. 12.13(d)
```

The spurs in the skeleton were reduced using the command

```
>> s1 = bwmorph(s, 'spur', 8); % Fig. 12.13(e)
```

where we repeated the operation 8 times, which in this case is equal approximately to one-half the value of `sig`. Several small spurs still remain in the skeleton. However, applying the previous function an additional 7 times (to complete the value of `sig`) yielded the result in Fig. 12.13(f), which is a reasonable skeleton representation of the input. As a rule of thumb, the value of `sig` of a Gaussian smoothing mask is a good guideline for the selection of the number of times a spur removal algorithm is applied. ■

12.3 Boundary Descriptors

In this section we discuss a number of *descriptors* that are useful when working with region boundaries. As will become evident shortly, many of these descriptors are applicable to regions also, and the grouping of descriptors in the toolbox does not make a distinction regarding their applicability. Therefore, some of the concepts introduced here are mentioned again in Section 12.4 when we discuss regional descriptors.

Descriptors also are called *features*.

12.3.1 Some Simple Descriptors

The *length* of a boundary is one of its simplest descriptors. The length of a 4-connected boundary is defined as the number of pixels in the boundary, minus 1. If the boundary is 8-connected, we count vertical and horizontal transitions as 1, and diagonal transitions as $\sqrt{2}$. (This descriptor can be computed using function `regionprops` discussed in Section 12.4.)

We extract the boundary of objects contained in image `f` using function `bwperim`, introduced in Section 12.1.1:

```
g = bwperim(f, conn)
```

where `g` is a binary image containing the boundaries of the objects in `f`. For 2-D connectivity, which is our focus, `conn` can have the values 4 or 8, depending on whether 4- or 8-connectivity (the default) is desired (see the margin note in Example 12.3 concerning the interpretation of these connectivity values). The objects in `f` can have any pixel values consistent with the image class, but all background pixels have to be 0. By definition, the perimeter pixels are nonzero and are connected to at least one other nonzero pixel.

The *diameter* of a boundary is defined as the Euclidean distance between the two points on the boundary that are farthest apart. These points are not always unique, as in a circle or square, but the assumption is that if the diameter is to be a useful descriptor, it is best applied to boundaries with a single pair of farthest points.† The line segment connecting these points is called the *major axis* of the boundary. The *minor axis* of a boundary is defined as the line perpendicular to the major axis, and of such length that a box passing through the outer four points of intersection of the boundary with the two axes completely encloses the boundary. This box is called the *basic rectangle*, and the ratio of the major to the minor axis is called the *eccentricity* of the boundary.

Custom function `diameter` (see Appendix C for a listing) computes the diameter, major axis, minor axis, and basic rectangle of a boundary *or* region. Its syntax is

`diameter`

```
s = diameter(L)
```

where `L` is a label matrix (Section 10.4) and `s` is a structure with the following fields:

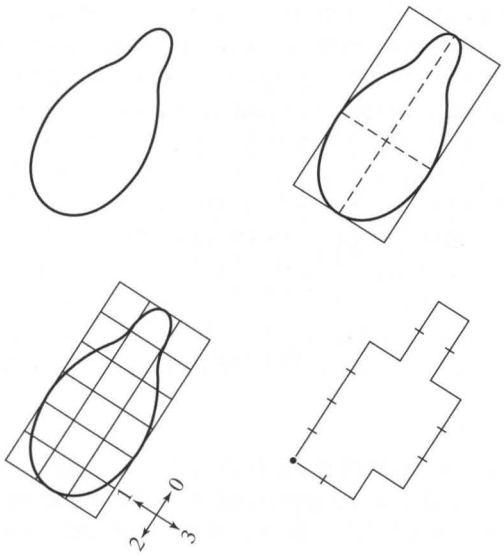
`s.Diameter`	A scalar, the maximum distance between any two pixels in the boundary or region.
`s.MajorAxis`	A 2×2 matrix, the rows of which contain the row and column coordinates for the endpoints of the major axis of the boundary or region.
`s.MinorAxis`	A 2×2 matrix, the rows of which contain the row and column coordinates for the endpoints of the minor axis of the boundary or region.
`s.BasicRectangle`	A 4×2 matrix, where each row contains the row and column coordinates of a corner of the basic rectangle.

12.3.2 Shape Numbers

The *shape number* of a boundary, generally based on 4-directional Freeman chain codes (see Section 12.2.1), is defined as the first difference of smallest magnitude (Bribiesca and Guzman [1980], Bribiesca [1981]). The *order* of a shape number is defined as the number of digits in its representation. Thus, the shape number of a boundary is given by parameter `c.diffmm` in function `fchcode` discussed in Section 12.2.1, and the order of the shape number is given by `length(c.diffmm)`.

As noted in Section 12.2.1, 4-directional Freeman chain codes can be made insensitive to the starting point by using the integer of minimum magnitude, and made insensitive to rotations that are multiples of 90° by using the first difference of the code. Thus, shape numbers are insensitive to the starting point and to rotations that are multiples of 90°. An approach used to normalize for arbitrary rotations is illustrated in Fig. 12.14. The procedure is to align one of

†When more than one pair of farthest points exist, they should be near each other and be dominant factors in determining boundary shape in order for their to be meaningful in the context of this discussion.



Chain code: 0 0 0 0 3 0 0 3 2 2 3 2 2 2 2 1 2 1 1
Difference: 3 0 0 0 3 1 0 3 3 0 1 3 0 0 3 1 3 0
Shape no.: 0 0 0 3 1 0 3 3 0 1 3 0 0 3 1 3 0 3

a b
c d

FIGURE 12.14
Steps in the generation of a shape number.

the coordinate axes with the major axis and then extract the 4-code based on the rotated figure. The x-axis can be aligned with the major axis of a region or boundary by using custom function `x2majoraxis` (see Appendix C). The syntax of this function is:

```
[C, theta] = x2majoraxis(A, B)
```

`x2majoraxis`

Here, `A = s.MajorAxis` is from function `diameter`, and `B` is an input (binary) image or boundary list. (As before, we assume that a boundary is a connected, closed curve.) Output `C` has the same form as the input (i.e., a binary image or a coordinate sequence. Because of possible round-off error, rotations can result in a disconnected boundary sequence, so postprocessing to relink the points (using, for example, `bwmorph` or `connectpoly`) may be necessary.

The tools required to implement an M-function that calculates shape numbers have been discussed already. They consist of function `bwboundaries` to extract the boundary, function `diameter` to find the major axis, function `bsubsamp` to reduce the resolution of the sampling grid, and function `fchcode` to extract the 4-directional Freeman code.

12.3.3 Fourier Descriptors

Figure 12.15 shows a K-point digital boundary in the xy-plane. Starting at an arbitrary point, (x_0, y_0), coordinate pairs $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{K-1}, y_{K-1})$ are encountered in traversing the boundary, say, in the counterclockwise direction.

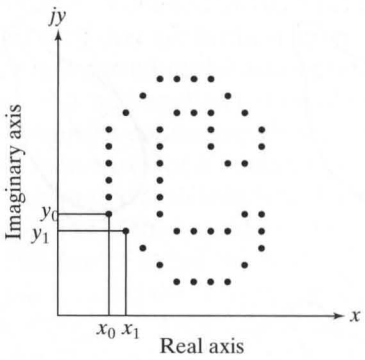


FIGURE 12.15 A digital boundary and its representation as a complex sequence. Point (x_0, y_0) (selected arbitrarily) is the starting point. Point (x_1, y_1) is the next counterclockwise point in the sequence.

These coordinates can be expressed in the form $x(k) = x_k$ and $y(k) = y_k$. With this notation, the boundary itself can be represented as the sequence of coordinates $s(k) = [x(k), y(k)]$, for $k = 0, 1, 2, \ldots, K - 1$. Moreover, each coordinate pair can be treated as a complex number so that

$$s(k) = x(k) + jy(k)$$

With reference to Section 4.1, the discrete Fourier transform of the 1-D sequence $s(k)$ can be written as

$$a(u) = \sum_{k=0}^{K-1} s(k) e^{-j2\pi uk/K}$$

for $u = 0, 1, 2, \ldots, K - 1$. The complex coefficients $a(u)$ are called the *Fourier descriptors* of the boundary. The inverse Fourier transform of these coefficients restores $s(k)$. That is,

$$s(k) = \frac{1}{K} \sum_{u=0}^{K-1} a(u) e^{j2\pi uk/K}$$

for $k = 0, 1, 2, \ldots, K - 1$. Suppose, however, that instead of all the Fourier coefficients, we use only the first P coefficients in computing the inverse. This is equivalent to setting $a(u) = 0$ for $u > P - 1$ in the preceding equation for $a(u)$. The result is the following *approximation* to $s(k)$:

$$\hat{s}(k) = \frac{1}{P} \sum_{u=0}^{P-1} a(u) e^{j2\pi u/K}$$

for $k = 0, 1, 2, \ldots, K - 1$. Although only P terms are used to obtain each component of $\hat{s}(k)$, k still ranges from 0 to $K - 1$. That is, the *same* number of points exists in the approximate boundary, but not as many terms are used in the reconstruction of each point. Recall from Chapter 4 that high-frequency components account for fine detail, and low-frequency components determine global shape. Thus, loss of detail in the boundary increases as P decreases.

The following function, `frdescp`, computes the Fourier descriptors of a boundary, `s`. Similarly, given a set of Fourier descriptors, function `ifrdescp` computes the inverse using a specified number of descriptors, to yield a closed spatial curve.

frdescp

```
function z = frdescp(s)
%FRDESCP Computes Fourier descriptors.
%   Z = FRDESCP(S) computes the Fourier descriptors of S, which is an
%   np-by-2 sequence of ordered coordinates describing a boundary.
%
%   Due to symmetry considerations when working with inverse Fourier
%   descriptors based on fewer than np terms, the number of points
%   in S when computing the descriptors must be even. If the number
%   of points is odd, FRDESCP duplicates the end point and adds it at
%   the end of the sequence. If a different treatment is desired, the
%   the sequence must be processed externally so that it has an even
%   number of points.
%
%   See function IFRDESCP for computing the inverse descriptors.

% Preliminaries.
[np, nc] = size(s);
if nc ~= 2
   error('S must be of size np-by-2.');
end
if np/2 ~= round(np/2);
   s(end + 1, :) = s(end, :);
   np = np + 1;
end

% Create an alternating sequence of 1s and -1s for use in centering
% the transform.
x = 0:(np - 1);
m = ((-1) .^ x)';

% Multiply the input sequence by alternating 1s and -1s to center
% the transform.
s(:, 1) = m .* s(:, 1);
s(:, 2) = m .* s(:, 2);

% Convert coordinates to complex numbers.
s = s(:, 1) + i*s(:, 2);

% Compute the descriptors.
z = fft(s);
```

Function `ifrdescp` is as follows:

ifrdescp

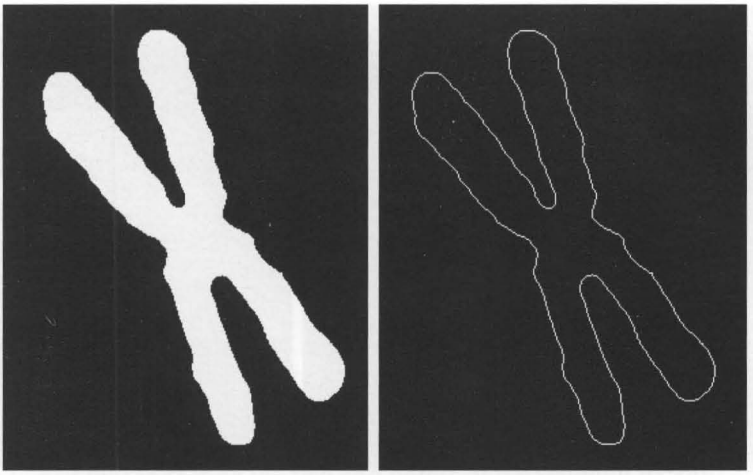
```
function s = ifrdescp(z, nd)
%IFRDESCP Computes inverse Fourier descriptors.
%   S = IFRDESCP(Z, ND) computes the inverse Fourier descriptors of
```

```
%   of Z, which is a sequence of Fourier descriptor obtained, for
%   example, by using function FRDESCP. ND is the number of
%   descriptors used to compute the inverse; ND must be an even
%   integer no greater than length(Z), and length(Z) must be even
%   also. If ND is omitted, it defaults to length(Z). The output,
%   S, is matrix of size length(Z)-by-2 containing the coordinates
%   of a closed boundary.

% Preliminaries.
np = length(z);
% Check inputs.
if nargin == 1
   nd = np;
end
if np/2 ~= round(np/2)
   error('length(z) must be an even integer.')
elseif nd/2 ~= round(nd/2)
   error('nd must be an even integer.')
end
% Create an alternating sequence of 1s and -1s for use in centering
% the transform.
x = 0:(np - 1);
m = ((-1) .^ x)';

% Use only nd descriptors in the inverse. Because the descriptors
% are centered, (np - nd)/2 terms from each end of the sequence are
% set to 0.
d = (np - nd)/2;
z(1:d) = 0;
z(np - d + 1:np) = 0;

% Compute the inverse and convert back to coordinates.
zz = ifft(z);
s(:, 1) = real(zz);
s(:, 2) = imag(zz);

% Multiply by alternating 1 and -1s to undo the earlier centering.
s(:, 1) = m .* s(:, 1);
s(:, 2) = m .* s(:, 2);
```

EXAMPLE 12.7: Fourier descriptors.

■ Figure 12.16(a) shows a binary image, `f`, similar to the one in Fig. 12.13(c), but obtained using a Gaussian mask of size 15×15 with `sigma = 9`, and thresholded at 0.7. The purpose was to generate an image that was not overly smooth in order to illustrate the effect that reducing the number of descriptors has on the shape of a boundary. The image in Fig. 12.16(b) was generated using the commands

```
>> b = bwboundaries(f, 'noholes');
```



a b

FIGURE 12.16
(a) Binary image. (b) Boundary extracted using function `bwboundaries`. The boundary has 1090 points.

```
>> b = b{1}; % There is only one boundary in this case.
>> bim = bound2im(b, size(f, 1), size(f, 2));
```

Figure 12.16(b) shows image `bim`. The boundary shown has 1090 points. Next, we computed the Fourier descriptors,

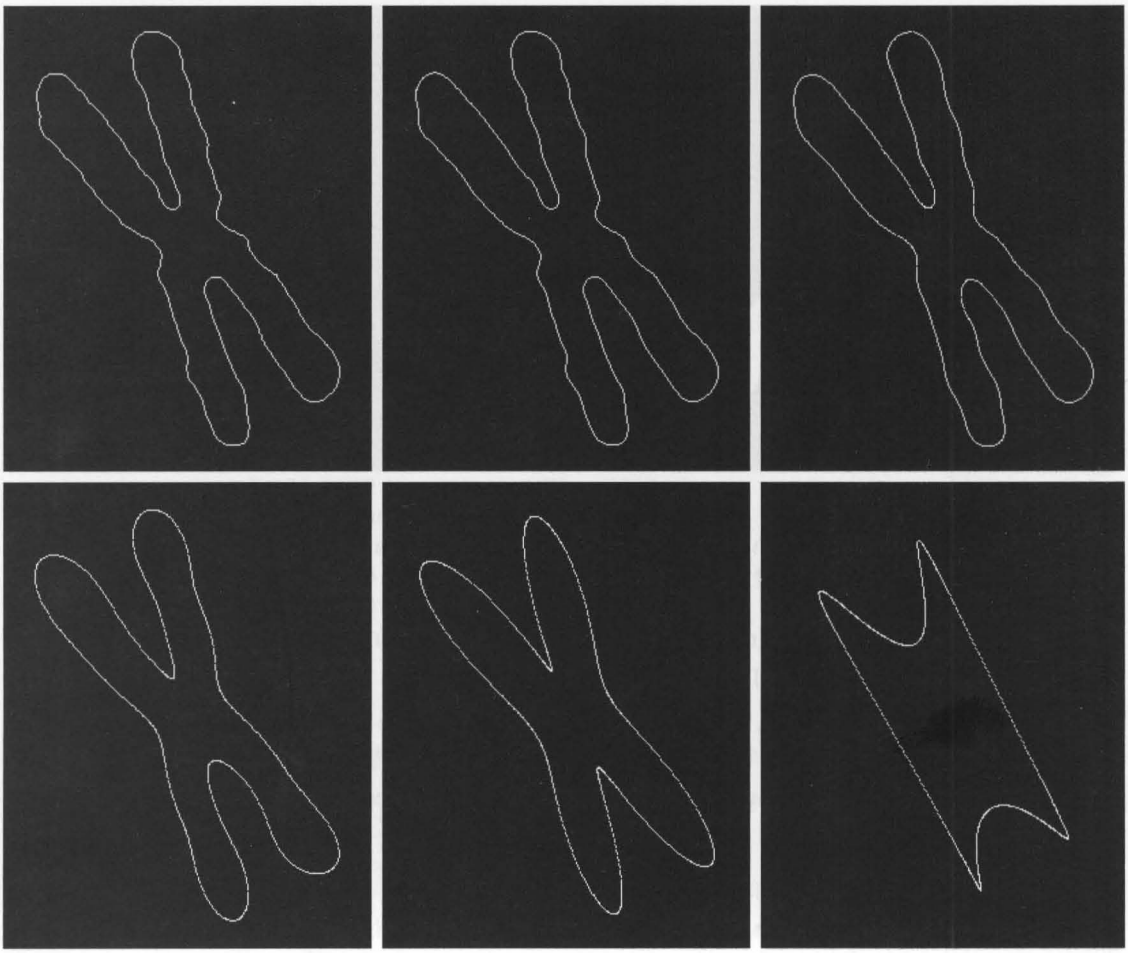
```
>> z = frdescp(b);
```

and obtained the inverse using approximately 50% of the possible 1090 descriptors:

```
>> s546 = ifrdescp(z, 546);
>> s546im = bound2im(s546, size(f, 1), size(f, 2));
```

Image `s546im` [Fig. 12.17(a)] shows close correspondence with the original boundary in Fig. 12.16(b). Some subtle details, such as a 1-pixel bay in the bottom-facing cusp in the original boundary, were lost but, for all practical purposes, the two boundaries are identical. Figures 12.17(b) through (f) show the results obtained using 110, 56, 28, 14, and 8 descriptors, which are approximately 10%, 5%, 2.5%, 1.25% and 0.7%, of the possible 1090 descriptors. The result obtained using 110 descriptors [Fig. 12.17(c)] shows slight further smoothing of the boundary, but, again, the general shape is quite close to the original. Figure 12.17(e) shows that even the result with 14 descriptors, a mere 1.25% of the total, retained the principal features of the boundary. Figure 12.17(f) shows distortion that is unacceptable because the main features of the boundary (the four long protrusions) were lost. Further reduction to 4 and 2 descriptors would result in an ellipse and, finally, a circle.

Some of the boundaries in Fig. 12.17 have one-pixel gaps due to round off in pixel values. These small gaps, common with Fourier descriptors, can be repaired with function `bwmorph` using the `'bridge'` option. ■



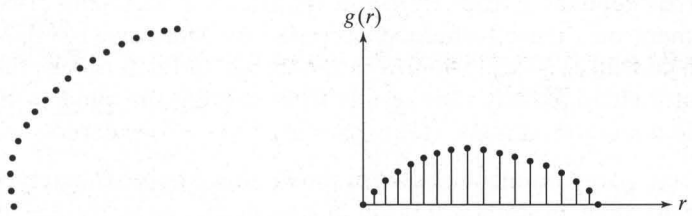
a b c
d e f

FIGURE 12.17 (a)–(f) Boundary reconstructed using 546, 110, 56, 28, 14, and 8 Fourier descriptors out of a possible 1090 descriptors.

As mentioned earlier, descriptors should be as insensitive as possible to translation, rotation, and scale changes. In cases where results depend on the order in which points are processed, an additional constraint is that descriptors should be insensitive to starting point. Fourier descriptors are not directly insensitive to these geometric changes, but the changes in these parameters can be related to simple transformations on the descriptors (see Gonzalez and Woods [2008]).

12.3.4 Statistical Moments

The shape of 1-D boundary representations (e.g., boundary segments and signature waveforms) can be described quantitatively by using statistical moments,



a b

FIGURE 12.18
(a) Boundary segment.
(b) Representation as a 1-D function.

such as the mean, variance, and higher-order moments. Consider Fig. 12.18(a), which shows a digital boundary segment, and Fig. 12.18(b), which shows the segment represented as a 1-D function, $g(r)$ of an arbitrary variable r. This function was obtained by connecting the two end points of the segment to form a "major" axis and then using function `x2majoraxis` discussed in Section 12.3.2 to align the major axis with the horizontal axis.

One approach for describing the shape of $g(r)$ is to normalize it to unit area and treat it as a histogram. In other words, $g(r_i)$ is treated as the probability of value r_i occurring. In this case, r is considered a random variable and the moments are

$$\mu_n = \sum_{i=0}^{K-1} (r_i - m)^n g(r_i)$$

where

$$m = \sum_{i=0}^{K-1} r_i g(r_i)$$

is the mean (average) value. Here, K is the number of boundary points, and μ_n is related to the shape of g. For example, the second moment, μ_2, measures the spread of the curve about the mean value of r and the third moment, μ_3, measures its symmetry with reference to the mean. Statistical moments are computed with function `statmoments` (see Section 5.2.4).

What we have accomplished is to reduce the description task to 1-D functions. The attractiveness of moments over other techniques is that their implementation is straightforward, and moments also carry a "physical" interpretation of boundary shape. The insensitivity of this approach to rotation is evident from Fig. 12.18. Size normalization can be achieved by scaling the range of values of g and r.

12.3.5 Corners

The boundary descriptors discussed thus far are global in nature. We conclude our discussion of boundary descriptors by developing two approaches for detecting *corners*, which are *local* boundary descriptors used widely in applications such as image tracking and object recognition. The following two methods are supported by the Image Processing Toolbox.

The Harris-Stephens corner detector

The Harris-Stephens corner detector (Harris and Stephens [1988]) is an improvement of a basic technique proposed by Moravec [1980]. Moravec's approach considers a local window in the image and determines the average change in image intensity that results from shifting the window by a small amount in various directions. Three cases need to be considered:

- If the image region encompassed by the window is approximately constant in intensity, then all shifts will result in a small change in average intensity.
- If the window straddles an edge, then a shift along the edge will result in a small change, but a shift perpendicular to the edge will result in a large change.
- If the windowed region contains a corner, then all shifts will result in a large change.[†] Thus, a corner can be detected by finding when the minimum change produced by any of the shifts is large (in terms of a specified threshold).

These concepts can be expressed mathematically as follows. Let $w(x, y)$ denote a spatial averaging (smoothing) mask in which all elements are nonnegative (i.e., a 3×3 mask whose coefficients are 1/9). Then, with reference to Sections 3.4 and 3.5, the average change in intensity, $E(x, y)$, at any coordinates (x, y) of an image $f(x, y)$ can be defined as

$$E(x,y) = \sum_s \sum_t w(s,t)\left[f(s+x,t+y) - f(s,t)\right]^2$$

where values of (s,t) are such that w and the image region corresponding to the expression in brackets overlap. By construction, we see that $E(x, y) \geq 0$.

Recall from basic mathematical analysis that the Taylor series expansion of a real function $f(s,t)$ about a point (x, y) is given by

$$f(s+x,t+y) = f(s,t) + \left[x\,\partial f(s,t)/\partial s + y\,\partial f(s,t)/\partial t\right] + \text{Higher-order terms}$$

For small shifts (i.e., small values of x and y), we can approximate this expansion using only the linear terms, in which case we can write E as

$$E(x,y) = \sum_s \sum_t w(s,t)\left[x\,\partial f(s,t)/\partial s + y\,\partial f(s,t)/\partial t\right]^2$$

The Harris-Stephens corner detector approximates the partial derivatives using the following spatial filtering with the masks $[-1\ 0\ 1]^T$ and $[-1\ 0\ 1]$:

$$f_s(s,t) = \partial f/\partial s = f(s,t) \star [-1\ 0\ 1]^T \text{ and } f_t(s,t) = \partial f/\partial t = f(s,t) \star [-1\ 0\ 1]$$

Then, we can write

[†] Certain types of noise, such as salt-and-pepper noise, can produce essentially the same response as a corner. However, the assumption when using this method is that the signal-to-noise ratio is large enough to allow reliable detection of corner features.

$$
\begin{aligned}
E(x,y) &= \sum_s \sum_t w(s,t)\left[xf_s(s,t) + yf_t(s,t)\right]^2 \\
&= \sum_s \sum_t w(s,t)x^2 f_s^2(s,t) + w(s,t)2xyf_s(s,t)f_t(s,t) + w(s,t)y^2 f_t^2(s,t) \\
&= x^2 \sum_s \sum_t w(s,t)f_s^2(s,t) + 2xy \sum_s \sum_t w(s,t)f_s(s,t)f_t(s,t) \\
&\qquad + y^2 \sum_s \sum_t w(s,t)f_t^2(s,t)
\end{aligned}
$$

The summation expressions in the preceding equation are correlations of the mask $w(x, y)$ with the terms shown (see Section 3.4), so we can write $E(x, y)$ as

$$E(x,y) = ax^2 + 2bxy + cy^2$$

where,

$$
\begin{aligned}
a &= w \star f_s^2 \\
b &= w \star f_s f_t \\
c &= w \star f_t^2
\end{aligned}
$$

We can express $E(x, y)$ in vector-matrix form, as follows,

$$E(x,y) = [x\ y]\mathbf{C}[x\ y]^T$$

where

$$\mathbf{C} = \begin{bmatrix} a & c \\ c & b \end{bmatrix}$$

The elements of this matrix are filtered (averaged) vertical and horizontal derivatives of the subimage area spanned by the averaging mask w.

Because $\mathbf{C}$ is symmetric, it can be diagonalized by a rotation of the coordinate axes (see the discussion at the end of Section 12.5):

$$\mathbf{C}_d = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

where λ_1 and λ_2 are the eigenvalues of $\mathbf{C}$, given by

$$\lambda_1, \lambda_2 = \frac{a+c}{2} \pm \left[\frac{4b^2 + (a-c)^2}{2}\right]^{\frac{1}{2}}$$

The Harris-Stephens corner detector is based on properties of these eigenvalues (note that $\lambda_1 \geq \lambda_2$).

Consult Noble and Daniel [1988] or any other text on basic matrices for a procedure used to obtain the eigenvalues of a matrix.

First, observe that both eigenvalues are proportional to the average value of local derivatives because of the way in which the elements of $\mathbf{C}$ were defined. In addition, both eigenvalues are nonnegative, for the following reason. As

stated earlier, $E(x, y) \geq 0$. Then $[x\ y]\mathbf{C}[x\ y]^T \geq 0$, which means that this quadratic form is positive semidefinite. This implies in turn that the eigenvalues of $\mathbf{C}$ are nonnegative. We can arrive at the same conclusion by noting, as you will see in Section 12.5, that the eigenvalues are proportional to the magnitude of the eigenvectors, which point in the direction of principal data spread. For example, in an area of constant intensity, both eigenvalues are zero. For a line one pixel thick, one eigenvalue will be 0 and the other positive. For any other type of configuration (including corners), both eigenvalues will be positive. These observations lead to the following conclusions based on *ideal* local image patterns:

(a) If the area encompassed by w is of constant intensity, then all derivatives are zero, $\mathbf{C}$ is the null matrix, and $\lambda_1 = \lambda_2 = 0$.
(b) If w contains an ideal black and white edge, then $\lambda_1 > 0$, $\lambda_2 = 0$, and the eigenvector associated with λ_1 is parallel to the image gradient.
(c) If w contains one corner of a black square on a white background (or vice versa) then there are two principal directions of data spread, and we have $\lambda_1 \geq \lambda_2 > 0$.

When working with real image data, we make less precise statements, such as, "if the area encompassed by w is nearly constant, then both eigenvalues will be small," and "if the area encompassed by w contains an edge, one eigenvalue will be large and the other small." Similarly, when dealing with corners, we look for the two eigenvalues being "large." Terms such as "small" and "large" are with respect to specified thresholds.

The key contribution made by Harris and Stephens was to use the concepts just presented to formalize and extend Moravec's original idea. Also, whereas Moravec used a constant averaging mask, Harris and Stephens proposed a Gaussian mask, which emphasizes the central part of the image under the mask:

$$w(s, t) = e^{-(s^2 + t^2)/2\sigma^2}$$

They also introduced the following response function

$$R = \text{Det} - k(\text{Tr})^2$$

where Det is the determinant of $\mathbf{C}$,

$$\text{Det} = \text{determinant}(\mathbf{C}) = \lambda_1\lambda_2 = ab - c^2$$

Tr is the trace of $\mathbf{C}$,

$$\text{Tr} = \text{trace}(\mathbf{C}) = \lambda_1 + \lambda_2 = a + b$$

and k is a *sensitivity parameter* (its range of values is discussed below). Using these results, we can express R directly in terms of a, b, and c:

$$R = ab - c^2 - k(a+b)^2$$

Using this formulation in terms of the elements a, b, and c has the slight advantage of not having to compute the eigenvalues directly for each displacement of the window.

Function R was constructed so that its value is low for flat areas, positive for corners, and negative for lines. The easiest way to demonstrate this is to expand R in terms of the eigenvalues:

$$R = (1 - 2k)\lambda_1\lambda_2 - k(\lambda_1^2 + \lambda_2^2)$$

Then, for example, considering the three ideal cases discussed earlier, you can see that, in a constant area both eigenvalues are 0 and, therefore, $R = 0$; in an area containing an edge one of the eigenvalues will be zero and, therefore, $R < 0$; for an ideal corner located symmetrically in the window, both eigenvalues will be equal and $R > 0$. These statements hold only if $0 < k < 0.25$ so, in the absence of additional information, this is a good range of values to choose for the sensitivity parameter.

The Harris-Stephens detector may be summarized as follows. We use MATLAB notation to emphasize the fact that the algorithm can be implemented using array operations:

1. Specify values for the parameter `k` and for the Gaussian smoothing function, `w`.
2. Compute the derivative images `fs` and `ft` by filtering the input image `f` using the filter masks `ws = [-1 0 1]'` and `wt = [-1 0 1]`, respectively. Obtain `fst = fs.*ft`.
3. Obtain arrays of coefficients `A`, `B`, and `C` by filtering `fs`, `ft`, and `fst`, respectively, with the averaging mask `w`. The respective elements of these arrays at any point are the a, b, c parameters defined earlier.
4. Compute the measure `R`:

```
R = (A.*B) - (C.^2) - k*(A + B).^2
```

We illustrate the performance of this detector in Example 12.8.

The minimum-eigenvalue corner detector

The method discussed in this section is based on property (c) discussed earlier. Assuming that the eigenvalues of $\mathbf{C}_d$ are ordered so that $\lambda_1 \geq \lambda_2$, the *minimum-eigenvalue corner detector* states that a corner has been found at the location of the center of the window over which the local derivatives were computed if

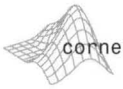
$$\lambda_2 > T$$

where T is a specified, nonnegative threshold, and λ_2 (the smallest eigenvalue) is computed using the analytical expression given earlier. Although this method clearly is a result of the Harris-Stephens development, it has gained acceptance as a rugged approach for corner detection in its own right. (e.g., see

Shi and Tomasi [1994], and Trucco and Verri [1998]). We illustrate both techniques in the following section.

Function `cornermetric`

The Harris-Stephens and minimum-eigenvalue detectors are implemented in the Image Processing Toolbox by function `cornermetric`,[†] with syntax




```
C = cornermetric(f, method, param1, val1, param2, val2)
```

where

- `f` is the input image.
- `method` can be either `'Harris'` or `'MinimumEigenvalue'`.
- `param1` is `'FilterCoefficients'`.
- `val1` is a vector containing the coefficients of a 1-D spatial filter mask, from which the function generates the corresponding 2-D square filter `w` discussed earlier. If `param1`, `val1` are not included in the call, the function generates a default 5×5 Gaussian filter using `fspecial('gaussian', [1 5], 1.5)` to generate the coefficients of the 1-D filter.
- `param2` is `'SensitivityFactor'`, applicable only to the Harris detector.
- `val2` is the value of the sensitivity factor k explained earlier. Its values are in the range $0 < k < 0.25$. The default value is 0.04.

The output of `cornermetric` is an array of the same size as the input image. The value of *each* point in the array is the corresponding metric R in the case of the Harris option, and the smallest eigenvector for the minimum-eigenvalue option. Our interest is in corners and, with either option, it is necessary to process the output (raw) array, `C`, further to determine which points are representative of valid corners, in terms of a specified threshold. We refer to points passing the threshold test as *corner points*. The following custom function (see Appendix C for the code) can be used for detecting these points:

`cornerprocess`

```
CP = cornerprocess(C, T, q)
```

where `C` is the output of `cornermetric`, `T` is a specified threshold, and `q` is the size of a square morphological structuring element used to reduce the number of corner points. That is, the corner points are dilated with a $q \times q$ structuring element of 1s to generate connected components. The connected components

[†] In the original paper by Harris and Stephens, the development starts with correlation, just as we did here, but the expressions for the derivatives and for computing a, b, and c are given in what may be interpreted ambiguously as convolution notation. The toolbox follows the notation in the paper and uses convolution also. As you will recall from Chapter 3, the difference between convolution and correlation is simply a rotation of the mask. The key point is that this does not affect the symmetry of $\mathbf{C}$ nor the form of the quadratic expression discussed earlier. Thus, the eigenvalues will be nonnegative using either convolution or correlation, and the result of the algorithm will be the same.

then are shrunk morphologically to single points. The actual reduction in the number of corner points depends on q and the proximity of the points.

EXAMPLE 12.8: Using functions `cornermetric` and `cornerprocess` to find corners in a gray-scale image.

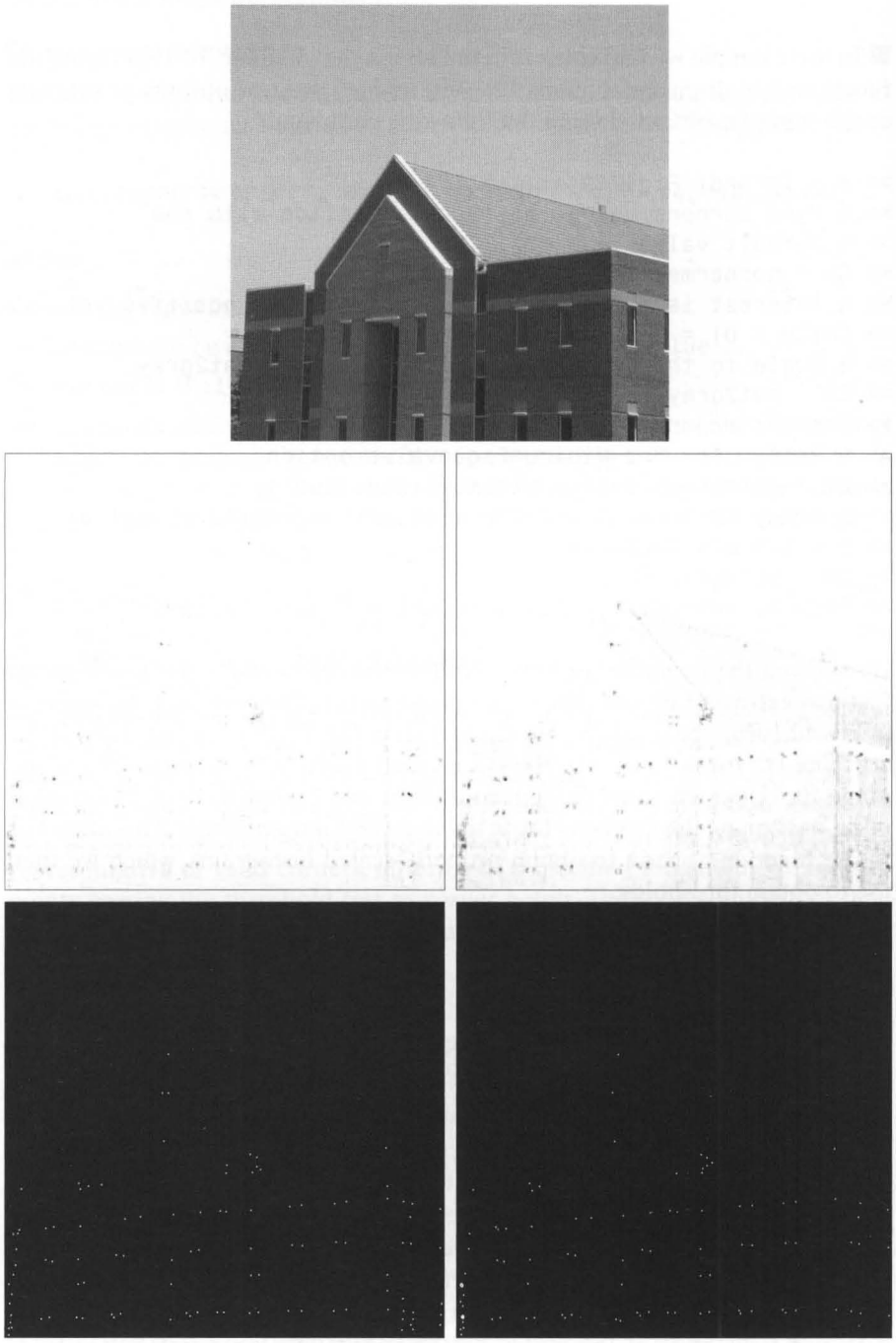
■ In this example we find corners in the image shown in Fig. 12.19(a) using the functions just discussed. Figures 12.19(b) and (c) are raw outputs of function `cornermetric`, obtained using the following commands:

```
>> f = imread('Fig1219(a).tif');
>> % Find corners using the 'Harris' option with the
>> % default values.
>> CH = cornermetric(f, 'Harris');
>> % Interest is in corners, so keep only the positive values.
>> CH(CH < 0) = 0;
>> % Scale to the range [0 1] using function mat2gray.
>> CH = mat2gray(CH);
>> imshow(imcomplement(CH)) % Figure 12.19(b).
>> % Repeat for the MinimumEigenvalue option.
>> CM = cornermetric(f, 'MinimumEigenvalue');
>> % Array CM consists of the smallest eigenvalues, all of
>> % which are positive.
>> CM = mat2gray(CM);
>> figure, imshow(imcomplement(CM)) % Figure 12.19(c).
```

We showed the negatives of Figs. 12.19(b) and (c) to make the low-contrast features extracted by `cornermetric` easier to see. Observe that the features in Fig. 12.19(b) are considerably dimmer than Fig. 12.19(c), a fact that can be attributed to using factor `k` in the Harris method. In addition to scaling to the range [0, 1] (which simplifies interpretation and comparison of the results), using `mat2gray` also converts the array to a valid image format. This allows us to use function `imhist` to obtain properly-scaled histograms, which we then use to obtain thresholds:

```
>> hH = imhist(CH);
>> hM = imhist(CM);
```

We used the percentile approach (see Section 11.3.5) to obtain the thresholds on which our definition of valid corners is based. The approach was to increase the percentile incrementally to generate thresholds for each corner detector and then process the image using function `cornerprocess` until the corners formed by the door frame and the front, right wall of the building disappeared. The largest threshold value before the corners disappeared was used as the value of `T`. The resulting percentiles were 99.45 and 99.70 for the Harris and minimum-eigenvalue methods, respectively. We used the corners just mentioned because they are good representations of image intensities between the dark and light parts of the building. Choosing other representative corners would give comparable results. The thresholds were computed as follows:



a
b c
d e

FIGURE 12.19
(a) Original image. (b) Raw output of the Harris, and (c) the minimum-eigenvalue detectors (shown as negative images to make low-contrast details easier to see; the borders are not part of the data). (d) and (e) Outputs of function `cornerprocess` using `q = 1` (the points were enlarged to make then easier to see).

```
>> TH = percentile2i(hH, 0.9945);
>> TM = percentile2i(hM, 0.9970);
```

Figures 12.19(d) and (e) were obtained using the commands

```
>> cpH = cornerprocess(CH, TH, 1); % Fig. 12.19(d).
>> cpM = cornerprocess(CM, TM, 1); % Fig. 12.19(e).
```

Each dot marks the center of window w where a valid corner point (designated by a 1-valued pixel) was detected. The correspondence of these points with respect to the image is easier to interpret by enclosing each point with, say, a circle, and superimposing the circles on the image [Figs. 12.20(a) and (b)]:

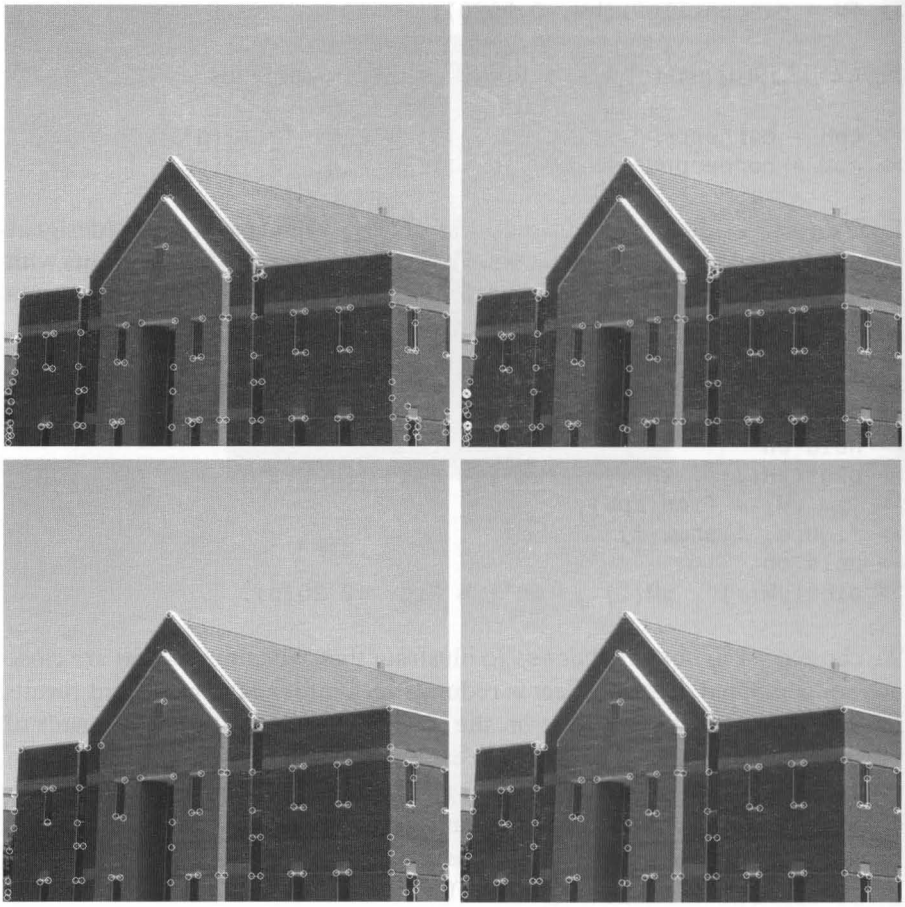
```
>> [xH yH] = find(cpH);
>> figure, imshow(f)
>> hold on
>> plot(yH(:)', xH(:)', 'wo') % Fig. 12.20(a).
>> [xM yM] = find(cpM);
>> figure, imshow(f)
>> hold on
>> plot(yM(:)', xM(:)', 'wo') % Fig. 12.20(b).
```

We chose `q = 1` in `cornerprocess` to illustrate that, when points that are close are not combined, the net effect is redundancy that leads to irrelevant results. For example, the heavy circles on the left of Fig. 12.20(b) are the result of numerous corner points being next to each other, caused primarily by random variations in intensity. Figures 12.20(c) and (d) show the results obtained with `q = 5` (the same size as the averaging mask) in function `cornerprocess`, and redoing the same sequence of steps used to generate Figs. 12.20(a) and (b). It is evident in these two images that the number of redundant corners was reduced significantly, thus giving a better description of the principal corners in the image.

Although the results are comparable, fewer false corners were detected using the minimum-eigenvalue method, which also has the advantage of having to be concerned with only one parameter (`T`), as supposed to two (`T` and `k`) with the Harris method. Unless the objective is to detect corners and lines simultaneously, the minimum-eigenvalue method typically is the preferred approach for corner detection. ■

12.4 Regional Descriptors

In this section we discuss a number of toolbox functions for region processing and introduce several additional functions for computing texture, moment invariants, and several other regional descriptors. Function `bwmorph` discussed in Section 10.3.4 is used frequently for the type of processing used in this section, as is function `roipoly` (Section 5.2.4).



a b
c d

FIGURE 12.20
(a) and (b) Corner points from Figs. 12.19(d) and (e), encircled and superimposed on the original image. (c) and (d) Corner points obtained using `q = 5` in function `cornerprocess`.

12.4.1 Function `regionprops`

Function `regionprops` is the toolbox's principal tool for computing region descriptors. This function has the syntax




```
D = regionprops(L, properties)
```

In addition to the measurements on binary images discussed here, function `regionprops` also computes several measurements for gray-scale images. Consult help for details.

where `L` is a label matrix (see Section 12.1.1) and `D` is a structure of length `max(L(:))`. The fields of the structure denote different measurements for each region, as specified by `properties`. Argument `properties` can be a comma-separated list of strings, a cell array containing strings, the single string `'all'`, or the string `'basic'`. Table 12.1 lists the set of valid property strings. If `properties` is the string `'all'`, then all the descriptors in Table 12.1 are computed. If `properties` is not specified or if it is the string `'basic'`, then the descriptors computed are `'Area'`, `'Centroid'`, and `'BoundingBox'`.

TABLE 12.1 Regional descriptors computed by function `regionprops`.

Valid strings for `properties`	Explanation
`'Area'`	The number of pixels in a region.
`'BoundingBox'`	1×4 vector defining the smallest rectangle containing a region. `BoundingBox` is defined by `[ul_corner width]`, where `ul_corner` is in the form `[x y]` and specifies the upper-left corner of the bounding box, and `width` is in the form `[x_width y_width]` and specifies the width of the bounding box along each dimension.
`'Centroid'`	1×2 vector; the center of mass of the region. The first element of `Centroid` is the horizontal coordinate of the center of mass, and the second is the vertical coordinate.
`'ConvexArea'`	Scalar; the number of pixels in `'ConvexImage'` (see below).
`'ConvexHull'`	$nv \times 2$ matrix; the smallest convex polygon that can contain the region. Each row of the matrix contains the horizontal and vertical coordinates of one of the nv vertices of the polygon.
`'ConvexImage'`	Binary image; the convex hull, with all pixels within the hull filled in (i.e., set to `on`). (For pixels on the boundary of the convex hull, `regionprops` uses the same logic as `roipoly` to determine whether a pixel is inside or outside the hull.)
`'Eccentricity'`	Scalar; the eccentricity of the ellipse that has the same second moments as the region. The eccentricity is the ratio of the distance between the foci of the ellipse and its major axis† length. The value is between 0 and 1, with 0 and 1 being degenerate cases (an ellipse whose eccentricity is 0 is a circle, while an ellipse with an eccentricity of 1 is a line segment).
`'EquivDiameter'`	Scalar; the diameter of a circle with the same area as the region. Computed as `sqrt(4*Area/pi)`.
`'EulerNumber'`	Scalar; the number of objects in the region minus the number of holes in those objects.
`'Extent'`	Scalar; the proportion of the pixels in the bounding box that are also in the region. Computed as `Area` divided by the area of the bounding box.
`'Extrema'`	8×2 matrix; the extremal points in the region. Each row of the matrix contains the horizontal and vertical coordinates of one of the points. The format of the eight rows is `[top-left, top-right, right-top, right-bottom, bottom-right, bottom-left, left-bottom, left-top]`.
`'FilledArea'`	The number of `on` pixels in `'FilledImage'`.
`'FilledImage'`	Binary image of the same size as the bounding box of the region. The `on` pixels correspond to the region, with all holes filled.
`'Image'`	Binary image of the same size as the bounding box of the region; the `on` pixels correspond to the region, and all other pixels are `off`.
`'MajorAxisLength'`	The length (in pixels) of the major axis† of the ellipse that has the same second moments as the region.
`'MinorAxisLength'`	The length (in pixels) of the minor axis† of the ellipse that has the same second moments as the region.
`'Orientation'`	The angle (in degrees) between the horizontal axis and the major axis† of the ellipse that has the same second moments as the region.
`'Perimeter'`	k-element vector containing the distance around the boundary of each of the k regions in the image.
`'PixelList'`	$np \times 2$ matrix whose rows are the `[horizontal vertical]` coordinates of the pixels in the region.
`'PixelIdxList'`	np-element vector containing the linear indices of the pixels in the region.
`'Solidity'`	Scalar; the proportion of the pixels in the convex hull that are also in the region. Computed as `Area/ConvexArea`.

† The use of *major* and *minor* axes in this context is different from the major and minor axes of the basic rectangle discussed in Section 12.3.1. For a discussion of moments of an ellipse, see Haralick and Shapiro [1992].

EXAMPLE 12.9: Using function `regionprops`.

■ To illustrate, we use `regionprops` to obtain the area and the bounding box for each region in an image `B`. We begin as follows:

```
>> B = bwlabel(B); % Convert B to a label matrix.
>> D = regionprops(B, 'area', 'boundingbox');
```

To extract the areas and number of regions we write

```
>> A = [D.Area];
>> NR = numel(A);
```

where the elements of vector `A` are the areas of the regions and `NR` is the number of regions. Similarly, we can obtain a single matrix whose rows are the bounding boxes of each region using the statement

```
V = cat(1, D.BoundingBox);
```

This array is of dimension `NR` $\times$ 4. ■

12.4.2 Texture

An important approach for describing a region is to quantify its texture content. In this section we illustrate the use of two custom functions and one toolbox function for computing texture based on statistical and spectral measures.

Statistical Approaches

An approach used frequently for texture analysis is based on statistical properties of the intensity histogram. One class of such measures is based on statistical moments of intensity values. As discussed in Section 5.2.4, the expression for the nth moment about the mean is given by

$$\mu_n = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i)$$

where z is a random variable indicating intensity, $p(z)$ is the histogram of the intensity levels in a region, L is the number of possible intensity levels, and

$$m = \sum_{i=0}^{L-1} z_i p(z_i)$$

is the mean (average) intensity. These moments can be computed with function `statmoments` discussed in Section 5.2.4. Table 12.2 lists some common descriptors based on statistical moments and also on uniformity and entropy. Keep in mind that the second moment, μ_2, is the variance, σ^2.

Custom function `statxture`, (see Appendix C) computes the texture measures in Table 12.2. Its syntax is

TABLE 12.2
Descriptors of texture based on intensity histograms.

Moment	Expression	Measure of Texture
Mean	$m = \sum_{i=0}^{L-1} z_i p(z_i)$	A measure of average intensity.
Standard deviation	$\sigma = \sqrt{\mu_2} = \sqrt{\sigma^2}$	A measure of average contrast.
Smoothness	$R = 1 - 1/(1 + \sigma^2)$	Measures the relative smoothness of the intensity in a region. R is 0 for a region of constant intensity and approaches 1 for regions with large excursions in the values of its intensity levels. In practice, the variance, σ^2, used in this measure is normalized to the range [0, 1] by dividing it by $(L-1)^2$.
Third moment	$\mu_3 = \sum_{i=0}^{L-1} (z_i - m)^3 p(z_i)$	Measures the skewness of a histogram. This measure is 0 for symmetric histograms; positive by histograms skewed to the right about the mean; and negative for histograms skewed to the left. Values of this measure are brought into a range of values comparable to the other five measures by dividing μ_3 by $(L-1)^2$, the same divisor we used to normalize the variance.
Uniformity	$U = \sum_{i=0}^{L-1} p^2(z_i)$	Measures uniformity. This measure is maximum when all intensity values are equal (maximally uniform) and decreases from there.
Entropy	$e = -\sum_{i=0}^{L-1} p(z_i) \log_2 p(z_i)$	A measure of randomness.

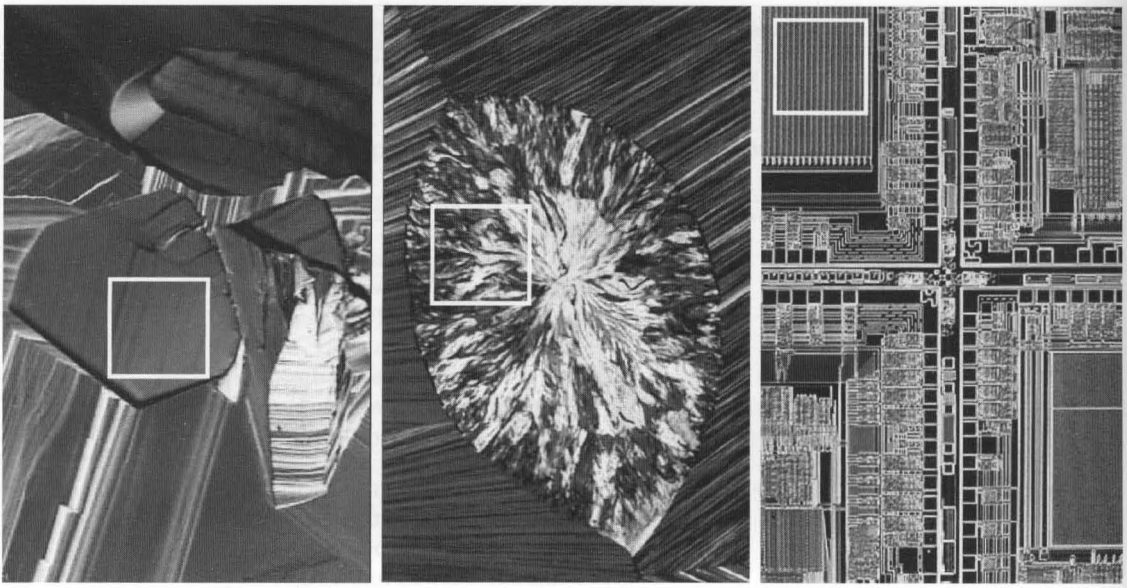
```
t = statxture(f, scale)
```

statxture

where `f` is an input image (or subimage) and `t` is a 6-element row vector whose components are the descriptors in Table 12.2, arranged in the same order. Parameter `scale` is a 6-element row vector also, whose components multiply the corresponding elements of `t` for scaling purposes. If omitted, `scale` defaults to all 1s.

EXAMPLE 12.10: Measures of statistical texture.

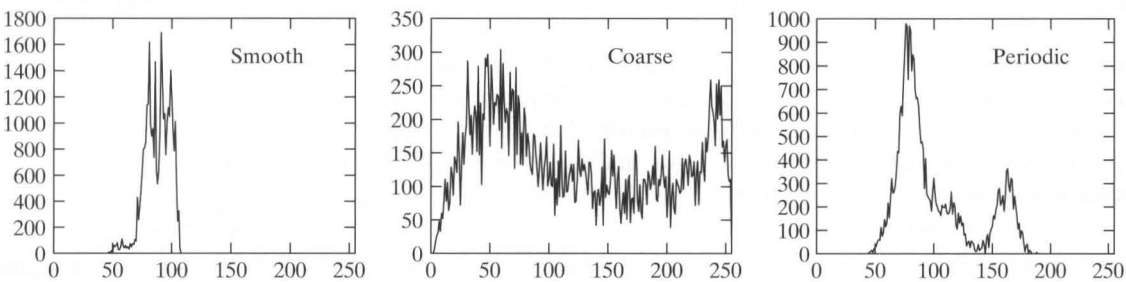
■ The three regions outlined by the white boxes in Fig. 12.21 are, from left to right, examples of smooth, coarse, and periodic texture. The histograms of these regions, obtained using function `imhist`, are shown in Fig. 12.22. The



a b c

FIGURE 12.21 The subimages in the white boxes from left to right are samples of smooth, coarse, and periodic texture. These are optical microscope images of a superconductor, human cholesterol, and a microprocessor. (Original images courtesy of Dr. Michael W. Davidson, Florida State University.)

entries in Table 12.3 were obtained by applying function `statxture` to each of the subimages in Fig. 12.21. These results are in general agreement with the texture content of their corresponding subimages. For example, the entropy of the coarse region [Fig. 12.21(b)] is higher than the others because the values of the pixels in that region are more random than the values in the other regions. This is true also for the contrast and for the average intensity in this case. On the other hand, this region is the least smooth and the least uniform, as indicated by the values of R and the uniformity measure. The histogram of the coarse region also shows the least symmetry with respect to the mean value, as



a b c

FIGURE 12.22 Histograms corresponding to the subimages in Fig. 12.21.

Texture	Average Intensity	Average Contrast	R	Third Moment	Uniformity	Entropy
Smooth	87.02	11.17	0.002	−0.011	0.028	5.367
Coarse	119.93	73.89	0.078	2.074	0.005	7.842
Periodic	98.48	33.50	0.017	0.557	0.014	6.517

TABLE 12.3
Texture measures for the regions enclosed by white squares in Fig. 12.21.

is evident in Fig. 12.22(b), and also by the largest value of the third moment in Table 12.3. ■

Measures of texture computed using only histograms carry no information regarding the relative position of pixels with respect to each other. This information is important when describing texture, and one way to incorporate it into texture analysis is to consider not only the distribution of intensities, but also the *relative positions* of pixels in an image.

Let O be an operator that defines the position of two pixels relative to each other, and consider an image, $f(x, y)$, with L possible intensity levels. Let $\mathbf{G}$ be a matrix whose element g_{ij} is the number of times that pixel pairs with intensities z_i and z_j occur in f in the position specified by O, where $1 \leq i, j \leq L$. A matrix formed in this manner is referred to as a *gray-level* (or *intensity*) *co-occurrence matrix*. Often, $\mathbf{G}$ is referred to simply as a *co-occurrence matrix*.

Figure 12.23 shows an example of how to construct a co-occurrence matrix using $L = 8$ and a position operator O defined as "one pixel immediately to the right." The array on the left in Fig. 12.23 is the image under consideration and the array on the right is matrix $\mathbf{G}$. We see that element $(1,1)$ of $\mathbf{G}$ is 1 because there is only one occurrence in f of a pixel valued 1 having a pixel valued 1 immediately to its right. Similarly, element $(6,2)$ of $\mathbf{G}$ is 3 because there are three occurrences in f of a pixel with value 6 having a pixel valued 2 immediately to its right. The other elements of $\mathbf{G}$ are computed in this manner. If we had

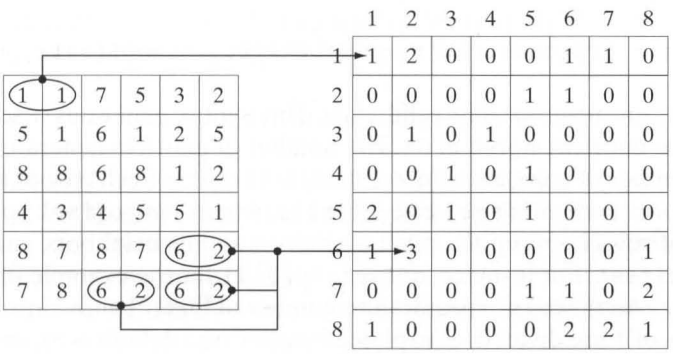


FIGURE 12.23
Generating a co-occurrence matrix.

defined O as, say, "one pixel to the right and one pixel above" then position (1,1) in $\mathbf{G}$ would have been 0 because there are no instances in f of a 1 with another 1 in the position specified by O. On the other hand, positions (1,3), (1,5), and (1,7) in $\mathbf{G}$ would all be 1s because intensity value 1 occurs in f with neighbors valued 3, 5, and 7 in the position specified by O, one time each.

The number of possible intensity levels in the image determines the size of matrix $\mathbf{G}$. For an 8-bit image (256 possible levels) $\mathbf{G}$ will be of size 256×256. This is not a problem when working with one matrix but, as you will see shortly, co-occurrence matrices sometimes are used in sequences, in which case the size of $\mathbf{G}$ is important from the point of view of computational loads. An approach used to reduce computations is to quantize the intensities into a few bands in order to keep the size of matrix $\mathbf{G}$ manageable. For example, in the case of 256 intensities we can do this by letting the first 32 intensity levels equal to 1, the next 32 equal to 2, and so on. This will result in a co-occurrence matrix of size 8×8.

The total number, n, of pixel pairs that satisfy O is equal to the *sum* of the elements of $\mathbf{G}$ ($n = 30$ in the preceding example). Then, the quantity

$$p_{ij} = \frac{g_{ij}}{n}$$

is an estimate of the probability that a pair of points satisfying O will have values (z_i, z_j). These probabilities are in the range [0, 1] and their sum is 1:

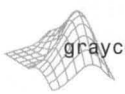
$$\sum_{i=1}^{K}\sum_{j=1}^{K} p_{ij} = 1$$

where K is the row (or column) dimension of square matrix $\mathbf{G}$. A *normalized* co-occurrence matrix is formed by dividing each of its terms by n:

$$\mathbf{G}_n = \frac{1}{n}\mathbf{G}$$

from which we see that each term of $\mathbf{G}_n$ is p_{ij}.

Function `graycomatrix` in the Image Processing Toolbox computes co-occurrence matrices. The syntax in which we are interested is




```
[GS, FS] = graycomatrix(f, 'NumLevels', n, 'Offset', offsets)
```

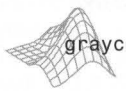
where `f` is an image of any valid class. This syntax generates a series of co-occurrence matrices stored in `GS`. The number of matrices generated depends on the number of rows in the `q` × `2` matrix, `offsets`. Each row of this matrix has the form `[row_offset, col_offset]`, where `row_offset` specifies the number of rows between the pixel of interest and its neighbors, and similarly for `col_offset`. For instance, `offsets = [0 1]` for the example in Fig. 12.23. Parameter `'NumLevels'` specifies the number of level "bands" into which the intensities of `f` are divided, as explained earlier (the default is 8), and `FS` is the resulting image, which is used by the function to generate `GS`. For example, we generate the co-occurence matrix in Fig. 12.23 as follows:

```
>> f = [1 1 7 5 3 2;
5 1 6 1 2 5;
8 8 6 8 1 2;
4 3 4 5 5 1;
8 7 8 7 6 2;
7 8 6 2 6 2];

>> f = mat2gray(f);
>> offsets = [0 1];
>> [GS, IS] = graycomatrix(f, 'NumLevels', 8, 'Offset',...
                           offsets)

GS =
    1    2    0    0    0    1    1    0
    0    0    0    0    1    1    0    0
    0    1    0    1    0    0    0    0
    0    0    1    0    1    0    0    0
    2    0    1    0    1    0    0    0
    1    3    0    0    0    0    0    1
    0    0    0    0    1    1    0    2
    1    0    0    0    0    2    2    1

IS =
    1    1    7    5    3    2
    5    1    6    1    2    5
    8    8    6    8    1    2
    4    3    4    5    5    1
    8    7    8    7    6    2
    7    8    6    2    6    2
```

Although the value of `NumLevels` needed to generate Fig. 12.23 is the same as the default, we showed it explicitly here for instructional purposes.

The way a co-occurrence matrix (or series of matrices) is used for texture description is based on the fact that, because **G** depends on O, the presence of intensity texture patterns may be detected by choosing appropriate position operators and analyzing the elements of the resulting **G**. The toolbox uses function `graycoprops` to generate descriptors:

```
stats = graycoprops(GS, properties)
```



where `stats` is a structure whose fields are the properties in Table 12.4. For example, if we specify `'Correlation'` or `'All'` for `properties`, then the field `stats.Correlation` gives the result of computing the correlation descriptor (we illustrate this in Example 12.11).

The quantities used in the correlation descriptor are as follows:

$$m_r = \sum_{i=1}^{K} iP(i)$$

$$m_c = \sum_{j=1}^{K} jP(j)$$

$$\sigma_r^2 = \sum_{i=1}^{K} (i - m_r)^2 P(i)$$

$$\sigma_c^2 = \sum_{j=1}^{K} (j - m_c)^2 P(j)$$

where

$$P(i) = \sum_{j=1}^{K} p_{ij} \quad \text{and} \quad P(j) = \sum_{i=1}^{K} p_{ij}$$

The quantity m_r is in the form of a mean computed along rows of $\mathbf{G}$ and m_c is a mean computed along the columns. Similarly, σ_r and σ_c are in the form of standard deviations computed along rows and columns respectively. Each of these terms is a scalar, independently of the size of $\mathbf{G}$.

TABLE 12.4 Properties supported by function `graycoprops`. The probability p_{ij} is the ij-th element of $\mathbf{G}/n$, where n is equal to the sum of the elements of $\mathbf{G}$.

Property	Description	Formula
`'Contrast'`	Returns a measure of the intensity contrast between a pixel and its neighbor over the entire image. Range = `[0 (size(G, 1) - 1) ^ 2]` `Contrast` is 0 for a constant image.	$\sum_{i=1}^{K}\sum_{j=1}^{K}(i-j)^2 p_{ij}$
`'Correlation'`	Returns a measure of how correlated a pixel is to its neighbor over the entire image. Range = `[-1 1]` `Correlation` is 1 or −1 for a perfectly positively or negatively correlated image, respectively. Correlation is `NaN` for a constant image.	$\sum_{i=1}^{K}\sum_{j=1}^{K}\frac{(i-m_r)(j-m_c)p_{ij}}{\sigma_r \sigma_c}$ $\sigma_r \neq 0; \sigma_c \neq 0$
`'Energy'`	Returns the sum of squared elements in `G`. Range = `[0 1]` `Energy` is 1 for a constant image.	$\sum_{i=1}^{K}\sum_{j=1}^{K} p_{ij}^2$
`'Homogeneity'`	Returns a value that measures the closeness of the distribution of elements in the `G` to the diagonal of `G`. Range = `[0 1]` `Homogeneity` is 1 for a diagonal `G`.	$\sum_{i=1}^{K}\sum_{j=1}^{K}\frac{p_{ij}}{1+\lvert i-j\rvert}$
`'All'`	Computes all the properties.	

Two additional measures that can be computed directly from the elements of $\mathbf{G}_n$ are the *maximum probability* (for measuring the strongest response of the co-occurrence matrix):

$$\text{maximum probability} = \max_{i,j}(p_{ij})$$

and the *entropy*, a measure of randomness, as before.

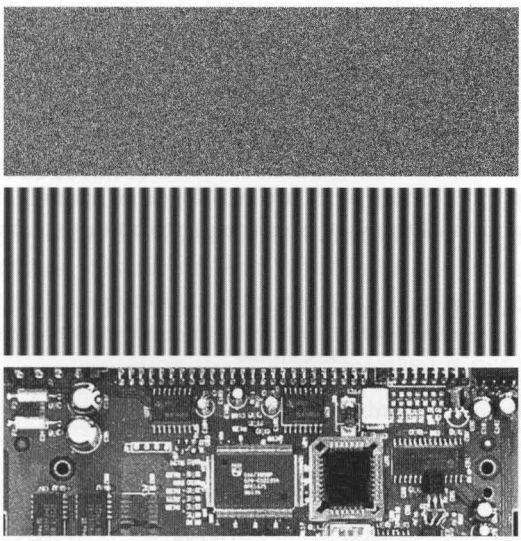
$$\text{entropy} = -\sum_{i=1}^{K}\sum_{j=1}^{K} p_{ij} \log_2 p_{ij}$$

EXAMPLE 12.11: Descriptors of texture based on co-occurrence matrices.

■ Figures 12.24(a)-(c) show images consisting of random, horizontally-periodic, and mixed textures, respectively. Our objectives in this example are to show (1) how to use individual co-occurrence matrices for texture description, and (2) how to use sequences of co-occurrence matrices for "discovering" texture patterns in an image. We illustrate the procedure for one image (the periodic texture) and list the results for the other two.

We begin by computing the co-occurrence matrix using the simplest, horizontal positional operator, `offsets = [0 1]`, which is the default (the texture patterns in which we are interested in this example are horizontal). We use all the number of levels (256 for `uint8` images) to get the finest possible differentiation in the descriptors:

```
>> f2 = imread('Fig1224(b).tif');
>> G2 = graycomatrix(f2,'NumLevels', 256);
>> G2n = G2/sum(G2(:)); % Normalized matrix.
>> stats2 = graycoprops(G2, 'all'); % Descriptors.
```



a
b
c

FIGURE 12.24 Images whose pixels exhibit (a) random, (b) periodic, and (c) mixed texture patterns. All images are of size 263 × 800 pixels.

Next we compute and list all the descriptors, including the two that we compute using the elements of G2n:

```
>> maxProbability2 = max(G2n(:));
>> contrast2 = stats2.Contrast;
>> corr2 = stats2.Correlation;
>> energy2 = stats2.Energy;
>> hom2 = stats2.Homogeneity;
>> for I = 1:size(G2n, 1);
      sumcols(I) = sum(-G2n(I,1:end).*log2(G2n(I,1:end)...
                                    + eps));
end
>> entropy2 = sum(sumcols);
```

The values of these descriptors are listed in the second row of Table 12.5. The other two rows were generated with the same procedure, using the other two images. The entries in this table agree with what one would expect from looking at the images in Fig. 12.24. For example, consider the Maximum Probability column in Table 12.5. The highest probability corresponds to the third co-occurrence matrix, which tells us that this matrix has the highest number of counts (largest number of pixel pairs occurring in the image relative to the positions in O) than the other two matrices. Examining Fig. 12.24(c) we see that there are large areas characterized by low variability in intensities in the horizontal direction, so we would expect the counts in $\mathbf{G}_3$ to be high.

The second column indicates that the highest correlation corresponds to $\mathbf{G}_2$. This tells us that the intensities in the second image are highly correlated. The repetitiveness of the periodic pattern in Fig. 12.24(b) reveals why this is so. Note that the correlation for $\mathbf{G}_1$ is essentially zero, indicating virtually no correlation between adjacent pixels, a characteristic of random images, such as the image in Fig. 12.24(a).The contrast descriptor is highest for $\mathbf{G}_1$ and lowest for $\mathbf{G}_2$. The less random an image is, the lowest its contrast tends to be. Although $\mathbf{G}_1$ has the lowest maximum probability, the other two matrices have many more zero or near zero probabilities. Keeping in mind that the sum of the values of a normalized co-occurrence matrix is 1, it is easy to see why the contrast descriptor tends to increase as a function of randomness.

The remaining three descriptors are explained in a similar manner. Energy increases as a function of the values of the probabilities squared. Thus the less

TABLE 12.5 Texture descriptors based on individual co-occurrence matrices for the image in Fig. 12.24.

Normalized Co-occurrence Matrix	Descriptor					
	Max Probability	Correlation	Contrast	Energy	Homogeneity	Entropy
$\mathbf{G}_1/n_1$	0.00006	−0.0005	10838	0.00002	0.0366	15.75
$\mathbf{G}_2/n_2$	0.01500	0.9650	570	0.01230	0.0824	6.43
$\mathbf{G}_3/n_3$	0.05894	0.9043	1044	0.00360	0.2005	13.63

randomness there is in a image, the highest the uniformity descriptor will be, as the fifth column in Table 12.5 shows. Homogeneity measures the concentration of values of **G** with respect to the main diagonal. The values of the denominator term in that descriptor are the same for all three co-occurrence matrices, and they decrease as i and j become closer in value (i.e., closer to the main diagonal). Thus, the matrix with the highest values of probabilities (numerator terms) near the main diagonal will have the highest value of homogeneity. Such a matrix corresponds to images with a rich gray-level content and areas of slowly varying intensity values. The entries in the sixth column of Table 12.5 are consistent with this interpretation.

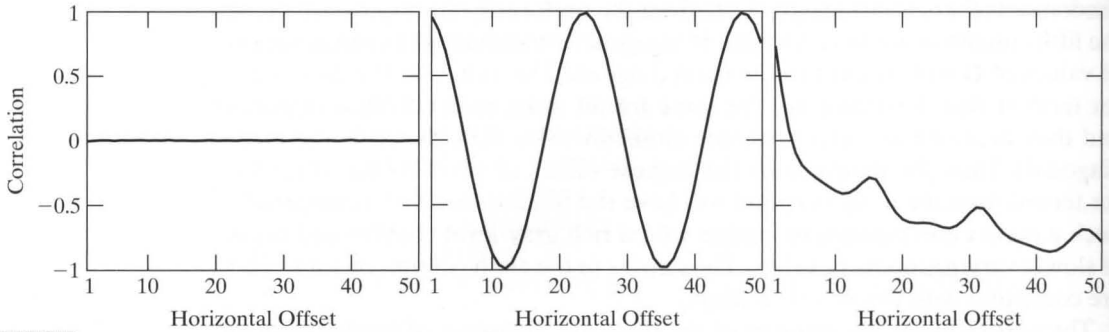
The entries in the last column of the table are measures of randomness in co-occurrence matrices, which in turn translate into measures of randomness in the corresponding images. As expected, $\mathbf{G}_1$ had the highest value because the image from which it was derived was totally random. The other two entries are self explanatory in this context.

Thus far we have dealt with single images and their co-occurrence matrices. Suppose that we want to "discover" (without looking at the images) if there are any sections in these images that contain repetitive components (i.e., periodic textures). One way to accomplish this goal is to examine the correlation descriptor for sequences of co-occurrence matrices, derived from these images by increasing the distance between neighbors. As mentioned earlier, it is customary when working with sequences of co-occurrence matrices to quantize the number of intensities in order to reduce matrix size and corresponding computational load. The following results were obtained using 8 levels, the default value. As before, we illustrate the procedure using the periodic image:

```
>> % Look at 50 increments of 1 pixel to the right.
>> offsets = [zeros(50,1) (1:50)']; %
>> G2 = graycomatrix(f2, 'Offset', offsets);
>> % G2 is of size 8-by-8-by-50.
>> stats2 = graycoprops(G2, 'Correlation');
>> % Plot the results.
>> figure, plot([stats2.Correlation]);
>> xlabel('Horizontal Offset')
>> ylabel('Correlation')
```

The other two images are processed in the same manner. Figure 12.25 shows plots of the correlation descriptors as a function of horizontal offset. Figure 12.25(a) shows that all correlation values are near 0, indicating that no correlation patterns were found in the random image. The shape of the correlation in Fig. 12.25(b) is a clear indication that the input image is periodic in the horizontal direction. Note that the correlation function starts at a high value and then decreases as the distance between neighbors increases, and then repeats itself.

Figure 12.25(c) shows that the correlation descriptor associated with the circuit board image decreases initially, but has a strong peak for an offset distance of 16 pixels. Analysis of the image in Fig. 12.24(c) shows that the upper solder



a b c

FIGURE 12.25 Values of the correlation descriptor as a function of horizontal offset (distance between adjacent pixels) corresponding to (a) the noisy, (b) the sinusoidal, and (c) the circuit board images in Fig. 12.24.

joints form a repetitive pattern approximately 16 pixels apart. The next major peak is at 32, caused by the same pattern. The amplitude of this peak is lower because the number of repetitions at this distance is less than at 16 pixels. A similar observation explains the even smaller peak at an offset of 48 pixels. ■

Spectral Measures of Texture

Spectral measures of texture are based on the Fourier spectrum, which is well-suited for describing the directionality of periodic or almost periodic 2-D patterns in an image. These global texture patterns, easily distinguishable as concentrations of high-energy bursts in the spectrum, generally are quite difficult to detect with spatial methods because of the local nature of these techniques. Thus, spectral texture is useful for discriminating between periodic and nonperiodic texture patterns, and, further, for quantifying differences between periodic patterns.

Interpretation of spectrum features is simplified by expressing the spectrum in polar coordinates to yield a function $S(r, \theta)$ where S is the spectrum function and r and θ are the independent variables in the polar coordinate system. For each direction θ, $S(r, \theta)$ is a 1-D function that can be written as $S_\theta(r)$. Similarly, for each frequency, r, $S(r, \theta)$ may be expressed as $S_r(\theta)$. Analyzing $S_\theta(r)$ for a fixed value of θ yields the behavior of the spectrum (such as the presence of peaks) along a radial direction from the origin, whereas analyzing $S_r(\theta)$ for a fixed value of r yields the behavior along a circle centered on the origin.

The *origin* in this discussion refers to the center of the frequency rectangle.

A global description is obtained by integrating (summing for discrete variables) these functions:

$$S(r) = \sum_{\theta=0}^{\pi} S_\theta(r)$$

and

$$S(\theta) = \sum_{r=1}^{R_0} S_r(\theta)$$

where R_0 is the radius of a circle centered at the origin.

The results of these two equations are a pair of values $[S(r), S(\theta)]$ for *each* pair of coordinates (r, θ). By varying these coordinates we can generate two 1-D functions, $S(r)$ and $S(\theta)$, that constitute a spectral-energy description of texture for an entire image or region. Furthermore, descriptors of these functions themselves can be computed in order to characterize their behavior quantitatively. Typical descriptors used for this purpose are the location of the highest value, the mean and variance of both the amplitude and axial variations, and the distance between the mean and the highest value of the function.

Function `specxture` (see Appendix C for the listing) can be used to compute the two preceding texture measurements. The syntax is

```
[srad, sang, S] = specxture(f)
```

`specxture`

where `srad` is $S(r)$, `sang` is $S(\theta)$, and `S` is the spectrum image (displayed using the log, as explained in Chapter 4).

EXAMPLE 12.12: Computing spectral texture.

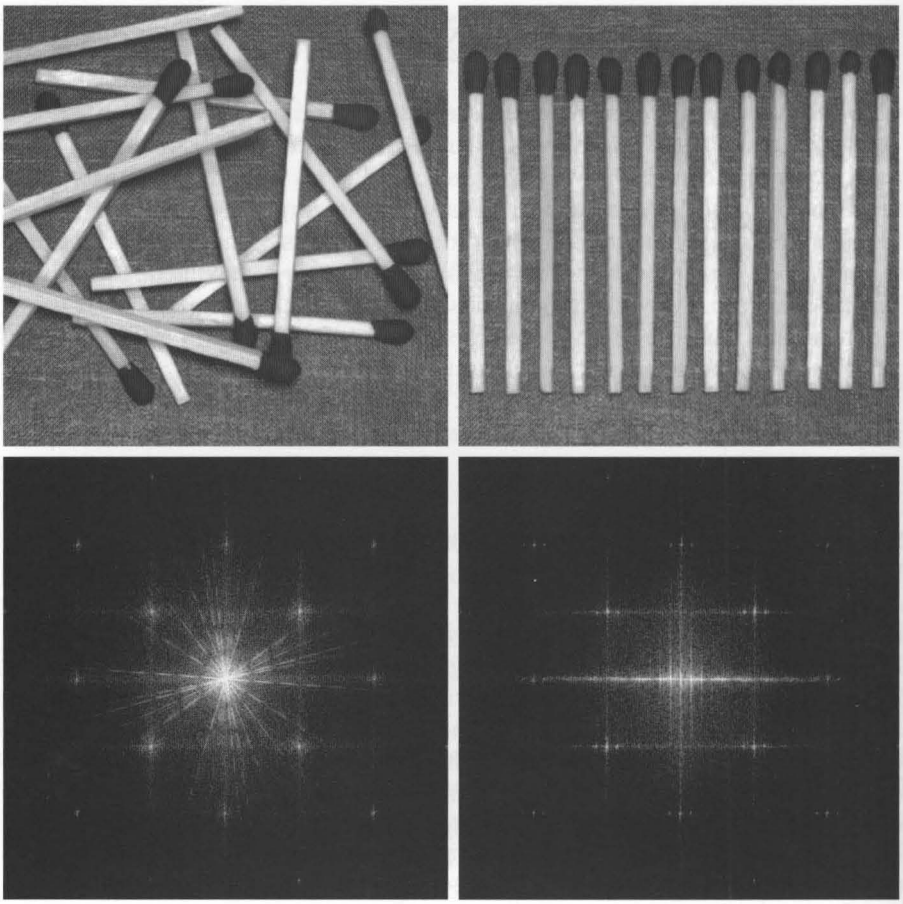
■ Figure 12.26(a) shows an image with randomly distributed objects and Fig. 12.26(b) shows an image containing the same objects, but arranged periodically. The corresponding Fourier spectra, computed using function `specxture`, are shown in Figs. 12.26(c) and (d). The periodic bursts of energy extending quadrilaterally in two dimensions in the Fourier spectra are due to the periodic texture of the coarse background material on which the matches rest. The other components of the spectra in Fig. 12.26(c) are caused by the random orientation of the strong edges in Fig. 12.26(a). By contrast, the main energy in Fig. 12.26(d) not associated with the background is along the horizontal axis, corresponding to the strong vertical edges in Fig. 12.26(b).

Figures 12.27(a) and (b) are plots of $S(r)$ and $S(\theta)$ for the random matches, and similarly in (c) and (d) for the ordered matches, all computed using function `specxture`. The plots were obtained with the commands `plot(srad)` and `plot(sang)`. The axes in Figs. 12.27(a) and (c) were scaled using

```
>> axis([horzmin horzmax vertmin vertmax])
```

discussed in Section 3.3.1, with the maximum and minimum values obtained from Fig. 12.27(a).

The plot of $S(r)$ corresponding to the randomly-arranged matches shows no strong periodic components (i.e., there are no peaks in the spectrum besides the peak at the origin, which is the dc component). On the other hand, the plot of $S(r)$ corresponding to the ordered matches shows a strong peak near $r = 15$ and a smaller one near $r = 25$. Similarly, the random nature of the energy bursts in Fig. 12.26(c) is quite apparent in the plot of $S(\theta)$ in Fig. 12.27(b). By contrast, the plot in Fig. 12.27(d) shows strong energy components in the region near the origin and at 90° and 180°. This is consistent with the energy distribution in Fig. 12.26(d). ■



a b
c d

FIGURE 12.26
(a) and (b) Images of unordered and ordered objects. (c) and (d) Corresponding spectra.

12.4.3 Moment Invariants

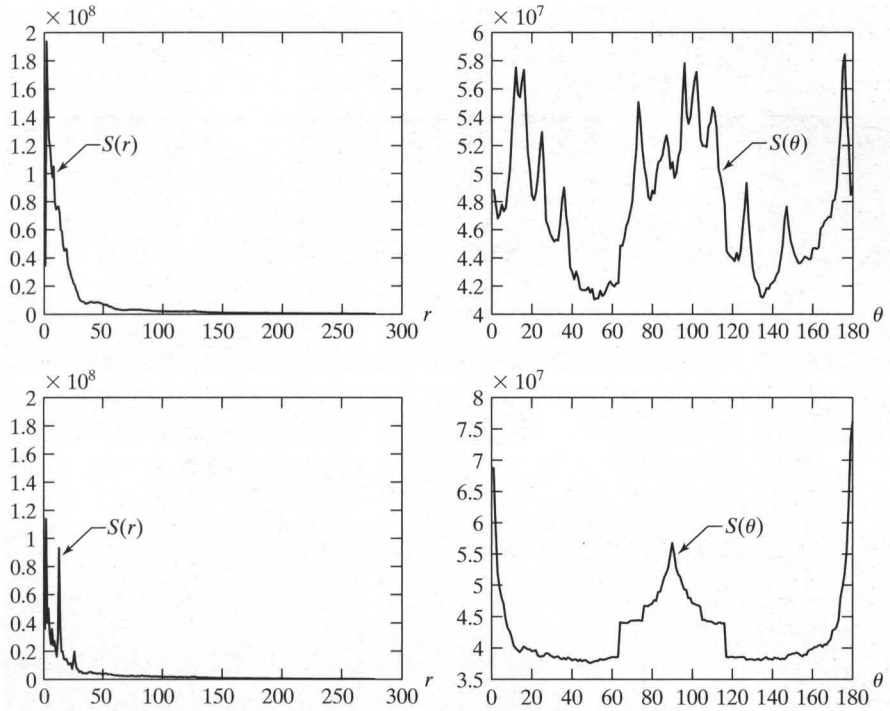
The 2-D *moment* of order $(p+q)$ of a digital image $f(x, y)$ of size $M \times N$ is defined as

$$m_{pq} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} x^p y^q f(x, y)$$

where $p = 0, 1, 2, \ldots$ and $q = 0, 1, 2, \ldots$ are integers. The corresponding *central moment* of order $(p+q)$ is defined as

$$\mu_{pq} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} (x - \bar{x})^p (y - \bar{y})^q f(x, y)$$

for $p = 0, 1, 2, \ldots$ and $q = 0, 1, 2, \ldots$, where



a b
c d

FIGURE 12.27
(a) and (b) Plots of $S(r)$ and $S(\theta)$ for the random image in Fig. 12.26(a). (c) and (d) Plots of $S(r)$ and $S(\theta)$ for the ordered image.

$$\bar{x} = \frac{m_{10}}{m_{00}} \quad \text{and} \quad \bar{y} = \frac{m_{01}}{m_{00}}$$

The *normalized central moment* of order $(p + q)$ is defined as

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}$$

where

$$\gamma = \frac{p + q}{2} + 1$$

for $p + q = 2, 3, \ldots$.

A set of seven 2-D *moment invariants* that are insensitive to translation, scale change, mirroring (to within a minus sign), and rotation can be derived from these equations.[†] They are listed in Table 12.6.

[†] Derivation of these results involves concepts that are beyond the scope of this discussion. The book by Bell [1965] and a paper by Hu [1962] contain detailed discussions of these concepts. For generating moment invariants of order higher than seven, see Flusser [2000]. Moment invariants can be generalized to n dimensions (see Mamistvalov [1998]).

TABLE 12.6
A set of seven moment invariants.

Moment order	Expression
1	$\phi_1 = \eta_{20} + \eta_{02}$
2	$\phi_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$
3	$\phi_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$
4	$\phi_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$
5	$\phi_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$
6	$\phi_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$
7	$\phi_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{30})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$

Custom M-function `invmoments` implements these seven equations. The syntax is as follows (see Appendix C for the code):

`invmoments`

```
phi = invmoments(f)
```

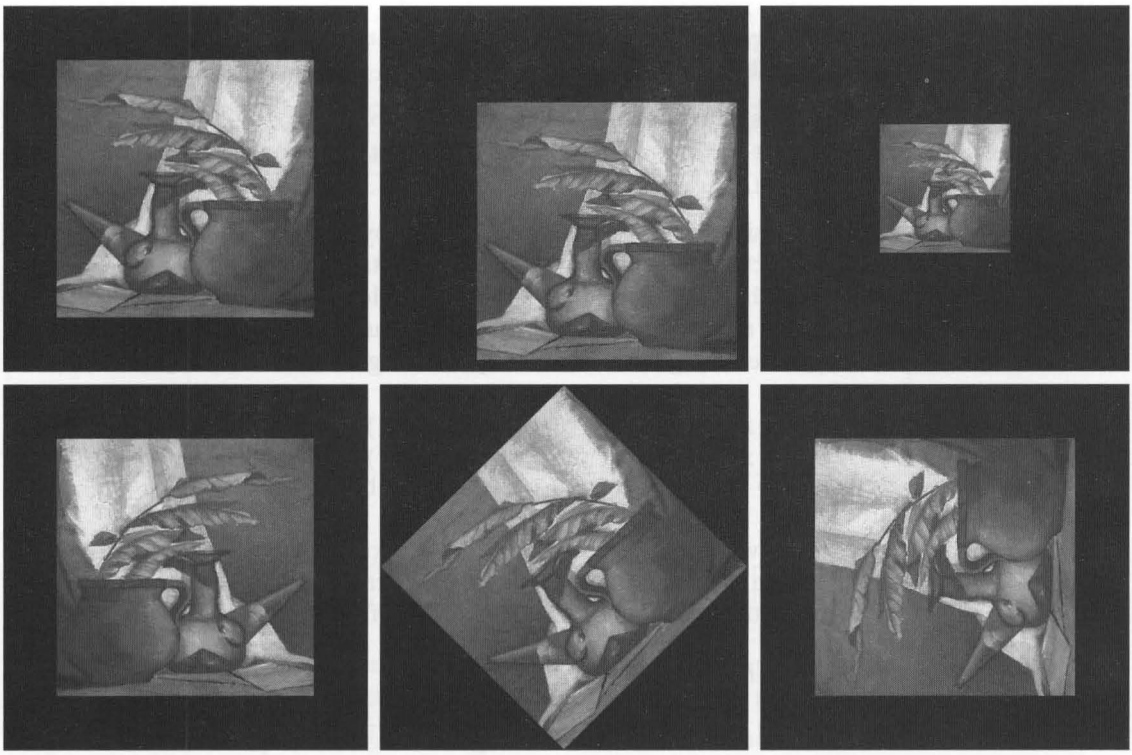
where `f` is the input image and `phi` is a seven-element row vector containing the moment invariants just defined.

EXAMPLE 12.13: Moment invariants.

■ The image in Fig. 12.28(a) was obtained from an original of size 400 × 400 pixels using the following commands:

```
>> f = imread('Fig1228(a).tif');
>> fp = padarray(f, [84 84], 'both'); % Padded for display.
```

This image was created using zero padding to make all displayed images consistent in size with the image occupying the largest area (568 × 568) which, as explained below, is the image rotated by 45°. The padding is for display purposes only, and was not used in moment computations. A translated image was created using the following commands:



a b c
d e f

FIGURE 12.28 (a) Original, padded image. (b) Translated image. (c) Half-size image. (d) Mirrored image. (e) Image rotated 45°. (f) Image rotated 90°.

```
>> ftrans = zeros(568, 568, 'uint8');
>> ftrans(151:550,151:550) = f;
```

A half-size and corresponding padded image were obtained using the commands

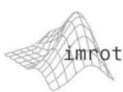
```
>> fhs = f(1:2:end, 1:2:end);
>> fhsp = padarray(fhs, [184 184], 'both');
```

A mirrored image was obtained using function `fliplr`:

```
>> fm = fliplr(f);
>> fmp = padarray(fm, [84 84], 'both'); % Padded for display.
```

To rotate the image we use function `imrotate`:

```
g = imrotate(f, angle, method, 'crop')
```



which rotates `f` by `angle` degrees in the counterclockwise direction. Parameter `method` can be one of the following:

- `'nearest'` uses nearest neighbor interpolation;
- `'bilinear'` uses bilinear interpolation (typically a good choice); and
- `'bicubic'` uses bicubic interpolation.

The image size is increased automatically by padding to fit the rotation. If `'crop'` is included in the argument, the central part of the rotated image is cropped to the same size as the original. The default is to specify `angle` only, in which case `'nearest'` interpolation is used and no cropping takes place.

The rotated images for our example were generated as follows:

```
>> fr45 = imrotate(f, 45, 'bilinear');
>> fr90 = imrotate(f, 90, 'bilinear');
>> fr90p = padarray(fr90, [84 84], 'both');
```

No padding was required in the first image because it is the largest image in the set. The 0s in `fr45` were generated automatically by `imrotate`.

We compute the moment invariants using function `invmoments`:

```
>> phi = invmoments(f);
```

are the moment invariants of the original image. Usually, the values of moment invariants are small and vary by several orders of magnitude, as you can see:

```
>> format short e
>> phi
phi =
    1.3610e-003   7.4724e-008   3.8821e-011   4.2244e-011
    4.3017e-022   1.1437e-014  -1.6561e-021
```

We bring these numbers into a range easier to analyze by reducing their dynamic range using a `log10` transformation. We also wish to preserve the sign of the original quantities:

```
>> format short
>> phinorm = -sign(phi).*(log10(abs(phi)))
phinorm =
    2.8662   7.1265  10.4109  10.3742   21.3674  13.9417  -20.7809
```

where `abs` was required because one of the numbers is negative. We preserved the sign of the original numbers by using `-sign(phi)`, where the minus sign was used because all numbers are fractions, thus giving a negative value when `log10` was computed. The central idea is that we are interested in the *invariance* of the numbers and not on their actual values. The sign needs to be

TABLE 12.7 The seven moment invariants of the images in Fig. 12.28. The values shown are for $-\text{sgn}(\phi_i)\log_{10}(|\phi_i|)$ to scale the numbers to a manageable range and simultaneously preserve the original sign of each moment.

Moment Invariant	Original Image	Translated	Half Size	Mirrored	Rotated 45°	Rotated 90°
ϕ_1	2.8662	2.8662	2.8664	2.8662	2.8661	2.8662
ϕ_2	7.1265	7.1265	7.1257	7.1265	7.1266	7.1265
ϕ_3	10.4109	10.4109	10.4047	10.4109	10.4115	10.4109
ϕ_4	10.3742	10.3742	10.3719	10.3742	10.3742	10.3742
ϕ_5	21.3674	21.3674	21.3924	21.3674	21.3663	21.3674
ϕ_6	13.9417	13.9417	13.9383	13.9417	13.9417	13.9417
ϕ_7	−20.7809	−20.7809	−20.7724	20.7809	−20.7813	−20.7809

preserved because it is used in ϕ_7 to detect if an image has been mirrored.

Using the preceding approach with all the images in Fig. 12.28 gave the results in Table 12.7. Observe how close the values are, indicating a high degree of invariance. This is remarkable, considering the variations in the images, especially in the half-size and rotated images with respect to the others. As expected, the sign of the mirrored image differed from all the others. ■

12.5 Using Principal Components for Description

Suppose that we have n spatially-registered images "stacked" in the arrangement shown in Fig. 12.29. There are n pixels for any given pair of coordinates (i, j), one pixel at that location for each image. These pixels can be arranged in the form of a column vector

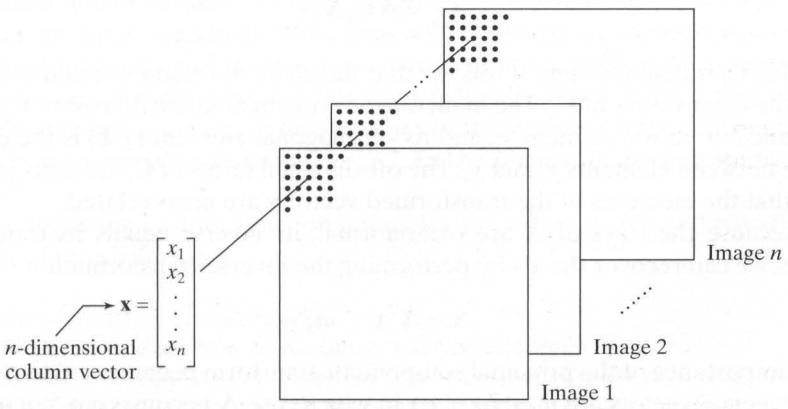


FIGURE 12.29 Forming a vector from corresponding pixels in a stack of images of the same size.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

If the images are of size $M \times N$ there will be total of MN such n-dimensional vectors comprising all pixels in the n images.

The mean vector, $\mathbf{m}_x$, of a vector population can be approximated by the sample average:

$$\mathbf{m}_x = \frac{1}{K}\sum_{k=1}^{K} \mathbf{x}_k$$

with $K = MN$. Similarly, the $n \times n$ covariance matrix, $\mathbf{C}_x$, of the population can be approximated by

$$\mathbf{C}_x = \frac{1}{K-1}\sum_{k=1}^{K} (\mathbf{x}_k - \mathbf{m}_x)(\mathbf{x}_k - \mathbf{m}_x)^T$$

where we use $K - 1$ instead of K to obtain an unbiased estimate of $\mathbf{C}_x$ from the samples.

The *principal components transform* (also called the *Hotelling transform*) is given by

$$\mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{m}_x)$$

The rows of matrix $\mathbf{A}$ are the eigenvectors of $\mathbf{C}_x$ normalized to unit length. Because $\mathbf{C}_x$ is real and symmetric, these vectors form an orthonormal set. It can be shown (Gonzalez and Woods [2008]) that

$$\mathbf{m}_y = \mathbf{0}$$

and that

$$\mathbf{C}_y = \mathbf{A}\mathbf{C}_x\mathbf{A}^T$$

Matrix $\mathbf{C}_y$ is diagonal, and it follows that the elements along its main diagonal are the eigenvalues of $\mathbf{C}_x$. The main diagonal element in the ith row of $\mathbf{C}_y$ is the variance of vector element y_i, and its off-diagonal element (j, k) is the covariance between elements y_j and y_k. The off-diagonal terms of $\mathbf{C}_y$ are zero, indicating that the elements of the transformed vector $\mathbf{y}$ are uncorrelated.

Because the rows of $\mathbf{A}$ are orthonormal, its inverse equals its transpose. Thus, we can recover the $\mathbf{x}$'s by performing the inverse transformation

$$\mathbf{x} = \mathbf{A}^T\mathbf{y} + \mathbf{m}_x$$

The importance of the principal components transform becomes evident when only q eigenvectors are used ($q < n$), in which case $\mathbf{A}$ becomes a $q \times n$ matrix, $\mathbf{A}_q$. Now the reconstruction is an *approximation*:

$$\hat{\mathbf{x}} = \mathbf{A}_q^T \mathbf{y} + \mathbf{m}_x$$

The mean square error between the exact and approximate reconstruction of the $\mathbf{x}$'s is given by the expression

$$e_{ms} = \sum_{j=1}^{n} \lambda_j - \sum_{j=1}^{q} \lambda_j$$
$$= \sum_{j=q+1}^{n} \lambda_j$$

The first line of this equation indicates that the error is zero if $q = n$ (that is, if all the eigenvectors are used in the inverse transformation). This equation also shows that the error can be minimized by selecting for $\mathbf{A}_q$ the q eigenvectors corresponding to the largest eigenvalues. Thus, the principal components transform is optimal in the sense that it minimizes the mean square error between the vectors $\mathbf{x}$ and their approximation $\hat{\mathbf{x}}$. The transform owes its name to using the eigenvectors corresponding to the largest (principal) eigenvalues of the covariance matrix. The example given later in this section further clarifies this concept.

A set of n registered images (each of size $M \times N$) is converted to a stack of the form shown in Fig. 12.29 by using the command:

```
>> S = cat(3, f1, f2,..., fn);
```

This image stack array, which is of size $M \times N \times n$ is converted to an array whose *rows* are n-dimensional vectors by using the following custom function (see Appendix C for the code):

```
[X, R] = imstack2vectors(S, MASK)
```

imstack2vectors

where S is the image stack and X is the array of vectors extracted from S using the approach in Fig. 12.29. Input MASK is an $M \times N$ logical or numeric array with nonzero elements in the locations where elements of S are to be used in forming X and 0s in locations to be ignored. For example, to use only vectors in the right, upper quadrant of the images in the stack, we set MASK to contain 1s in that quadrant and 0s elsewhere. The default for MASK is all 1s, meaning that all image locations are used to form X. Finally, R is a column vector that contains the *linear* indices of the locations of the vectors extracted from S. We show how to use MASK in Example 13.2. In the present discussion we use the default.

The following custom function computes the mean vector and covariance matrix of the vectors in X.

covmatrix

```
function [C, m] = covmatrix(X)
%COVMATRIX Computes the covariance matrix and mean vector.
%   [C, M] = COVMATRIX(X) computes the covariance matrix C and the
%   mean vector M of a vector population organized as the rows of
%   matrix X. This matrix is of size K-by-N, where K is the number
```

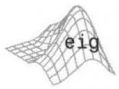
```
%   of samples and N is their dimensionality. C is of size N-by-N
%   and M is of size N-by-1. If the population contains a single
%   sample, this function outputs M = X and C as an N-by-N matrix of
%   NaN's because the definition of an unbiased estimate of the
%   covariance matrix divides by K - 1.

K = size(X, 1);
X = double(X);
% Compute an unbiased estimate of m.
m = sum(X, 1)/K;
% Subtract the mean from each row of X.
X = X - m(ones(K, 1), :);
% Compute an unbiased estimate of C. Note that the product is X'*X
% because the vectors are rows of X.
C = (X'*X)/(K - 1);
m = m'; % Convert to a column vector.
```

The following function implements the concepts developed thus far in this section. Note the use of structures to simplify the output arguments.

principalcomps

```
function P = principalcomps(X, q)
%PRINCIPALCOMPS Principal-component vectors and related quantities.
%   P = PRINCIPALCOMPS(X, Q) Computes the principal-component
%   vectors of the vector population contained in the rows of X, a
%   matrix of size K-by-n where K (assumed to be > 1)is the number
%   of vectors and n is their dimensionality. Q, with values in the
%   range [0, n], is the number of eigenvectors used in constructing
%   the principal-components transformation matrix. P is a structure
%   with the following fields:
%
%     P.Y      K-by-Q matrix whose columns are the principal-
%              component vectors.
%     P.A      Q-by-n principal components transformation matrix
%              whose rows are the Q eigenvectors of Cx corresponding
%              to the Q largest eigenvalues.
%     P.X      K-by-n matrix whose rows are the vectors
%              reconstructedfrom the principal-component vectors.
%              P.X and P.Y are identical if Q = n.
%     P.ems    The mean square error incurred in using only the Q
%              eigenvectors corresponding to the largest
%              eigenvalues. P.ems is 0 if Q = n.
%     P.Cx     The n-by-n covariance matrix of the population in X.
%     P.mx     The n-by-1 mean vector of the population in X.
%     P.Cy     The Q-by-Q covariance matrix of the population in
%              Y. The main diagonal contains the eigenvalues (in
%              descending order) corresponding to the Q
%              eigenvectors.

K = size(X. 1);
X = double(X);
```

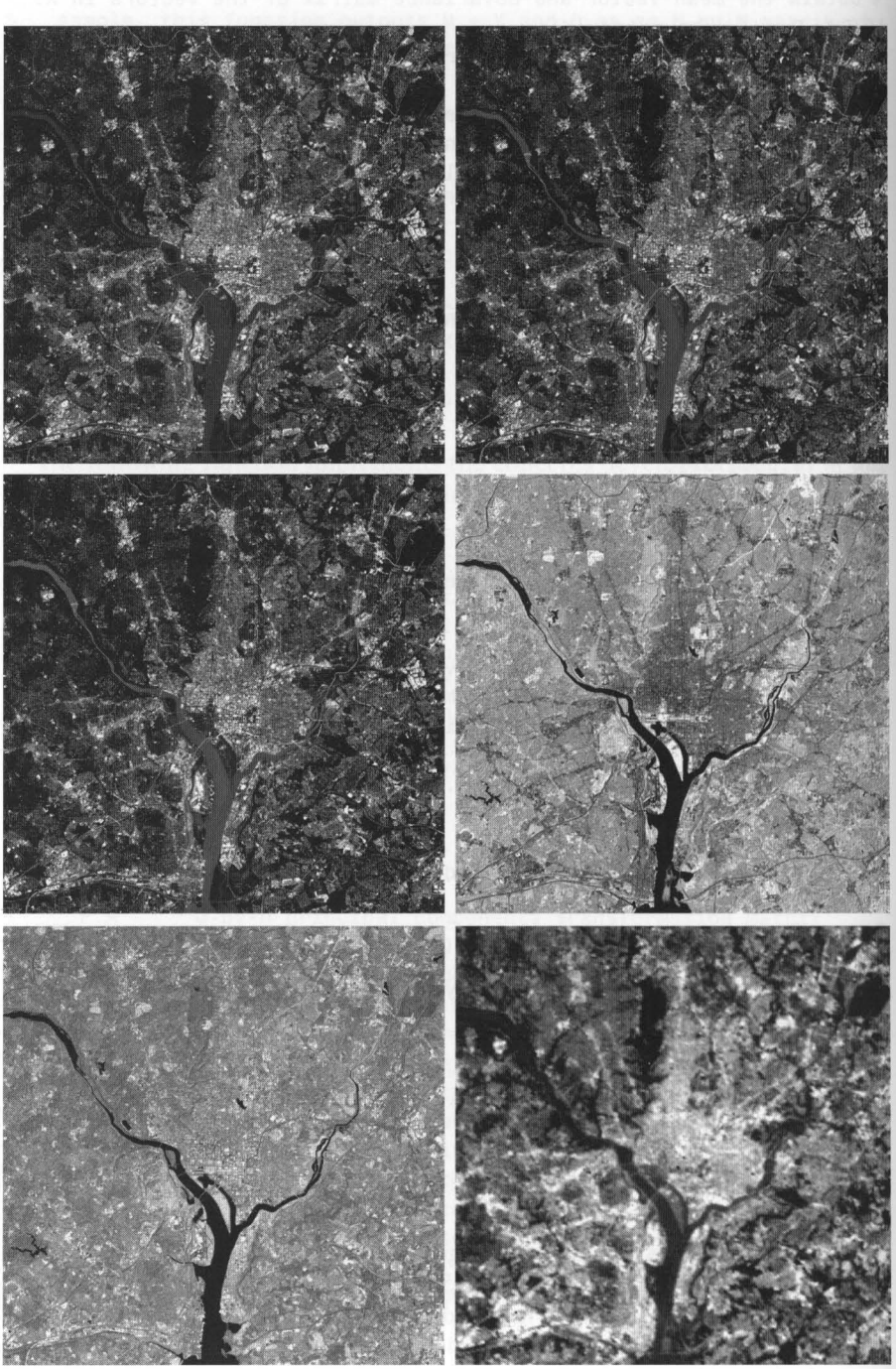
```
% Obtain the mean vector and covariance matrix of the vectors in X.
[P.Cx, P.mx] = covmatrix(X);
P.mx = P.mx'; % Convert mean vector to a row vector.

% Obtain the eigenvectors and corresponding eigenvalues of Cx.  The
% eigenvectors are the columns of n-by-n matrix V.  D is an n-by-n
% diagonal matrix whose elements along the main diagonal are the
% eigenvalues corresponding to the eigenvectors in V, so that X*V =
% D*V.
[V, D] = eig(P.Cx);

% Sort the eigenvalues in decreasing order.  Rearrange the
% eigenvectors to match.
d = diag(D);
[d, idx] = sort(d);
d = flipud(d);
idx = flipud(idx);
D = diag(d);
V = V(:, idx);

% Now form the q rows of A from the first q columns of V.
P.A = V(:, 1:q)';

% Compute the principal component vectors.
Mx = repmat(P.mx, K, 1); % M-by-n matrix.  Each row = P.mx.
P.Y = P.A*(X - Mx)'; % q-by-K matrix.

% Obtain the reconstructed vectors.
P.X = (P.A'*P.Y)' + Mx;

% Convert P.Y to a K-by-q array and P.mx to n-by-1 vector.
P.Y = P.Y';
P.mx = P.mx';

% The mean square error is given by the sum of all the
% eigenvalues minus the sum of the q largest eigenvalues.
d = diag(D);
P.ems = sum(d(q + 1:end));

% Covariance matrix of the Y's:
P.Cy = P.A*P.Cx*P.A';
```



`[V, D] = eig(A)` returns the eigenvectors of A as the columns of matrix V, and the corresponding eigenvalues along the main diagonal of diagonal matrix D.

EXAMPLE 12.14: Using principal components.

■ Figure 12.30 shows six satellite images of size 512 × 512 pixels, corresponding to six spectral bands: visible blue (450–520 nm), visible green (520–600 nm), visible red (630–690 nm), near infrared (760–900 nm), middle infrared (1550–1750 nm), and thermal infrared (10,400–12,500 nm). The objective of this example is to illustrate the use of function `principalcomps` for principal-components work. The first step is to organize the elements of the six images



a b
c d
e f

FIGURE 12.30 Six multispectral images in the (a) visible blue, (b) visible green (c) visible red, (d) near infrared, (e) middle infrared, and (f) thermal infrared bands. (Images courtesy of NASA.)

in a stack of size $512 \times 512 \times 6$ as discussed earlier:

```
>> S = cat(3, f1, f2, f3, f4, f5, f6);
```

where the `f`'s correspond to the six multispectral images just discussed. Then we organize the stack into array `X`:

```
>> X = imstack2vectors(S);
```

Next, we obtain the six principal-component images by using `q = 6` in function `principalcomps`:

```
>> P = principalcomps(X, 6);
```

The first component image is generated and displayed with the commands

```
>> g1 = P.Y(:, 1);
>> g1 = reshape(g1, 512, 512);
>> imshow(g1, [ ])
```

The other five images are obtained and displayed in the same manner. The eigenvalues are along the main diagonal of `P.Cy`, so we use

```
>> d = diag(P.Cy);
```

where `d` is a 6-dimensional column vector because we used `q = 6` in the function.

Figure 12.31 shows the six principal-component images just computed. The most obvious feature is that a significant portion of the contrast detail is contained in the first two images, and image contrast decreases rapidly from there. The reason can be explained by looking at the eigenvalues. As Table 12.8 shows, the first two eigenvalues are quite large in comparison with the others. Because the eigenvalues are the variances of the elements of the **y** vectors, and variance is a measure of contrast, it is not unexpected that the images corresponding to the dominant eigenvalues would exhibit significantly higher contrast.

Suppose that we use a smaller value of `q`, say `q = 2`. Then, reconstruction is based only on two principal component images. Using

Using a few component images to describe a larger set of images is a form of data compression.

```
>> P = principalcomps(X, 2);
```

and statements of the form

```
>> h1 = P.X(:, 1);
>> h1 = mat2gray(reshape(h1, 512, 512));
```

The values of `P.X(:, 1)` are outside the range [0, 1]. Using `mat2gray` scales the intensities of `h1` to this range.

for each image resulted in the reconstructed images in Fig. 12.32. Visually, these images are quite close to the originals in Fig. 12.30. In fact, even the differ-

a b
c d
e f

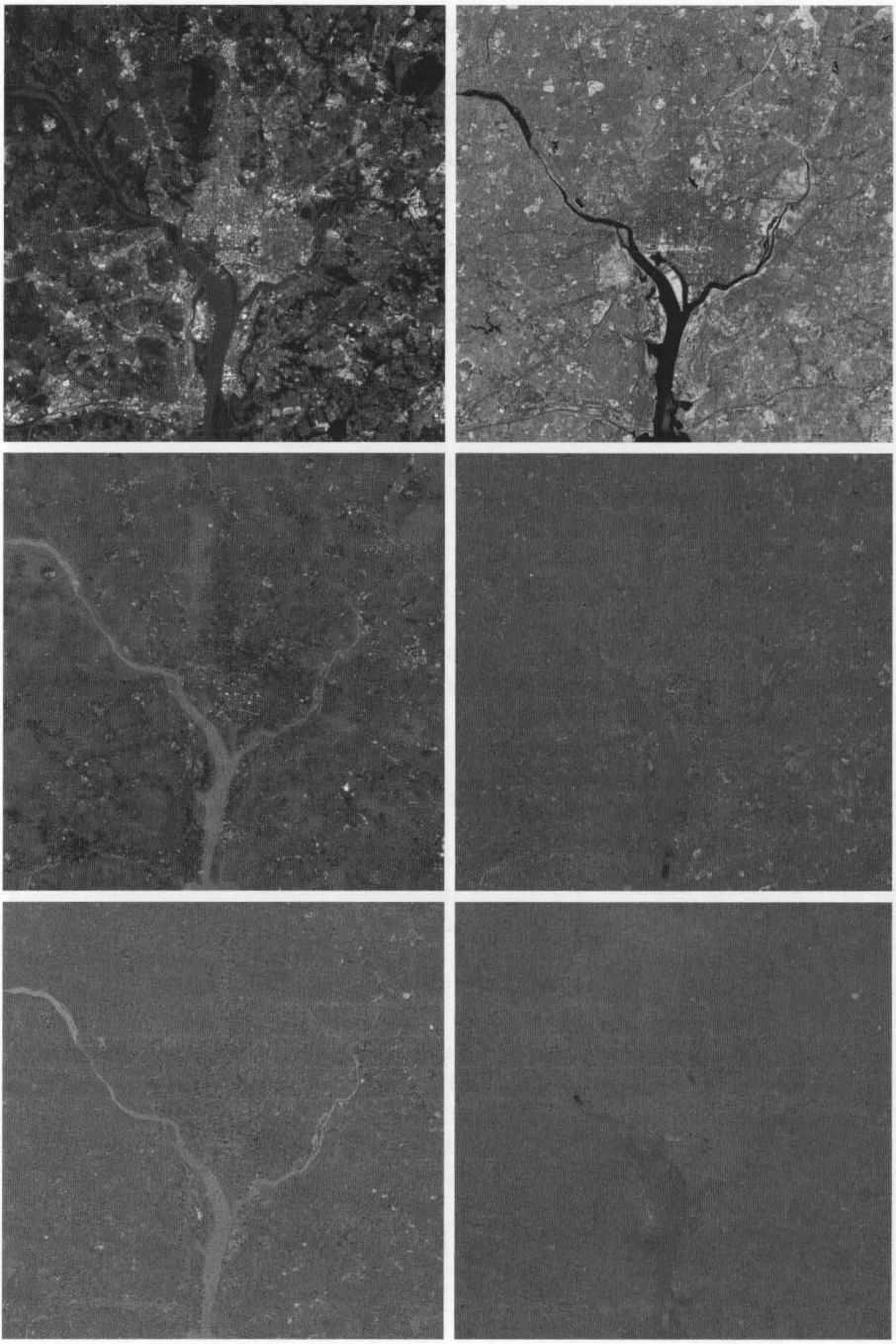
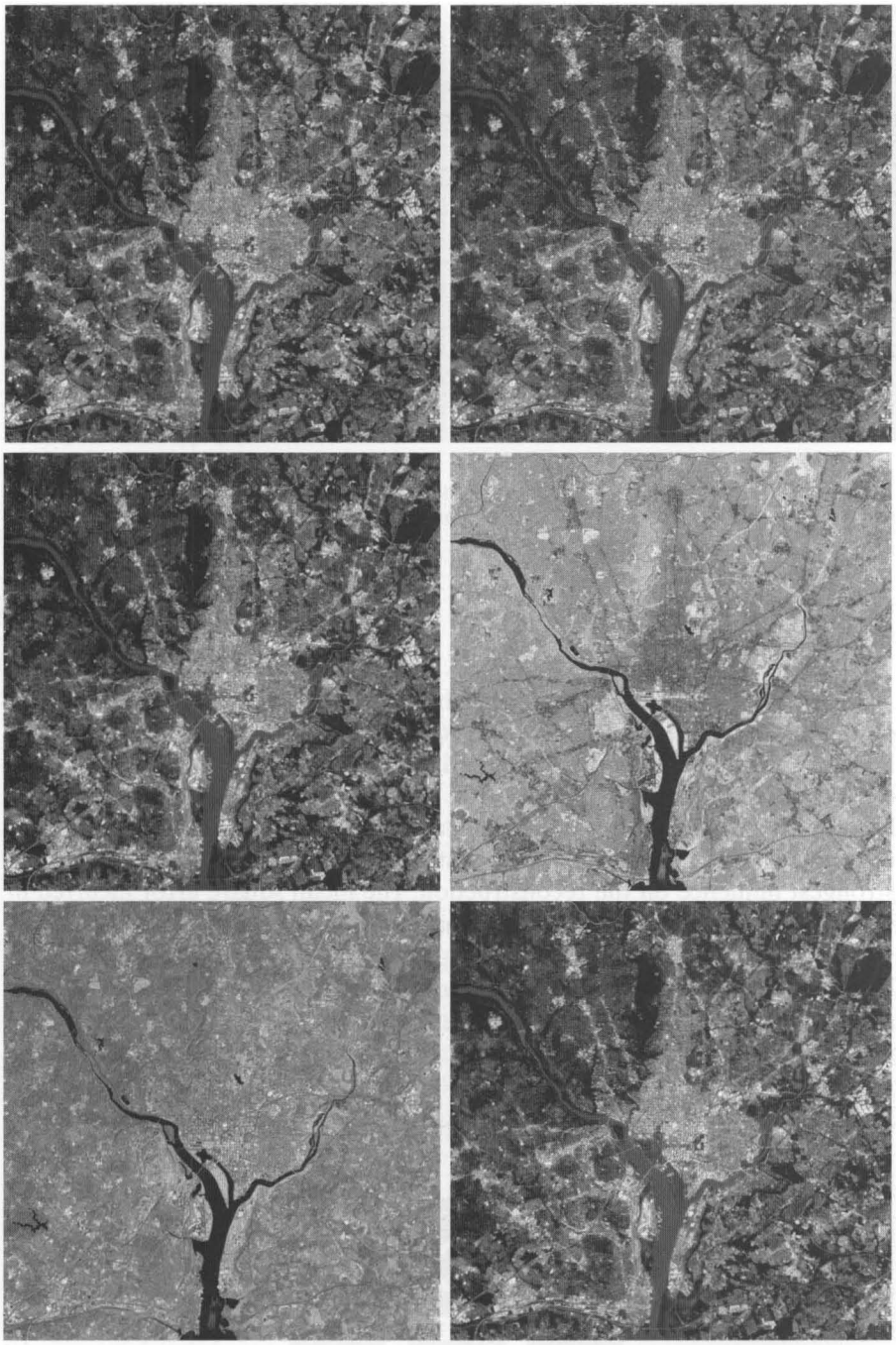


FIGURE 12.31 Principal-component images corresponding to the images in Fig. 12.30.

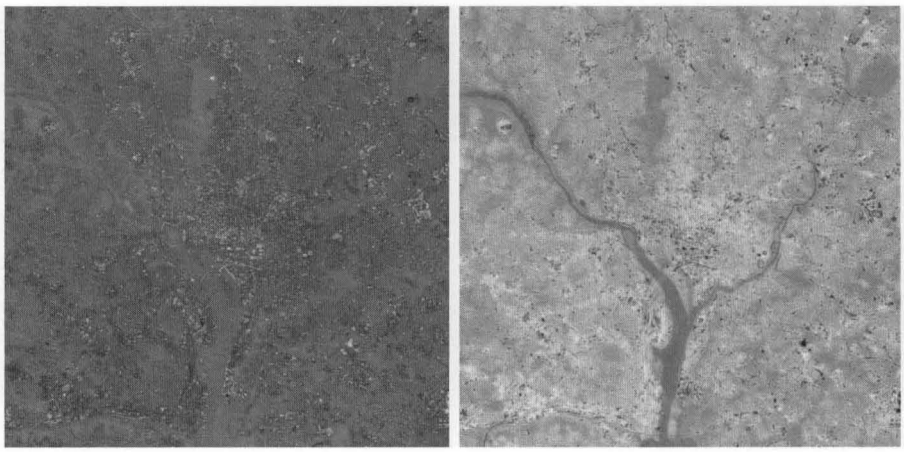
TABLE 12.8 Eigenvalues of `P.Cy` when `q = 6`.

λ_1	λ_2	λ_3	λ_4	λ_5	λ_6
10352	2959	1403	203	94	31



a b
c d
e f

FIGURE 12.32 Multispectral images reconstructed using only the two principal-component images with the largest variance. Compare with the originals in Fig. 12.30.



a b

FIGURE 12.33
(a) Difference between Figs. 12.30(a) and 12.32(a). (b) Difference between Figs. 12.30(f) and 12.32(f). Both images are scaled to the full [0, 255] 8-bit intensity scale.

ence images show little degradation. For instance, to compare the original and reconstructed band 1 images, we write

```
>> D1 = tofloat(f1) - h1;
>> imshow(D1, [])
```

Figure 12.33(a) shows the result. The low contrast in this image is an indication that little visual data was lost when only two principal component images were used to reconstruct the original image. Figure 12.33(b) shows the difference of the band 6 images. The difference here is more pronounced because the original band 6 image is actually blurry. But the two principal-component images used in the reconstruction are sharp, and they have the strongest influence on the reconstruction. The mean square error incurred in using only two principal component images is given by

```
P.ems
ans =
    1.7311e+003
```

which is the sum of the four smaller eigenvalues in Table 12.7. ■

Before leaving this section, we illustrate how function `principalcomps` can be used to align objects in the direction of the eigenvectors corresponding to the principal eigenvalues.[†] As noted earlier, eigenvalues are proportional to variance (spread of the data). By forming `X` from the 2-D *coordinates* of the objects, the basic idea of the approach is to align the objects spatially in the direction of their principal data spread. We illustrate the method with an example.

[†] See Gonzalez and Woods [2008] for more details on how to use principal components for 2-D data alignment.

EXAMPLE 12.15: Using principal components for object alignment.

■ The first row in Fig. 12.34 shows three images of characters oriented randomly. The objective in this example is to use principal components to align the characters vertically. This procedure is typical of techniques used to assess the orientation of objects in automated image analysis, thus simplifying subsequent object recognition tasks. In the following, we work out the details for Fig. 12.34(a). The remaining images are processed in the same way.

We begin by converting the data to binary form. That is, for the first image, we perform the following operation.

```
>> f = im2bw(imread('Fig1234(a).tif'));
```

The next step is to extract the coordinates of all the 1-valued pixels:

```
>> [x1 x2] = find(f);
```

Then, we form array `X` from these coordinates,

```
>> X = [x1 x2];
```

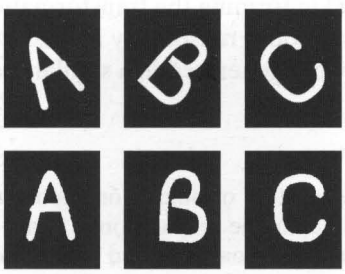
apply function `principalcomps`,

```
>> P = principalcomps(X, 2);
```

and transform the input coordinates into the output coordinates using the transformation matrix **A**:

```
>> A = P.A;
>> Y = (A*(X'))';
```

where the transposes shown are necessary because all elements of `X` are processed as a unit, unlike the original equation, which is stated in terms of a single vector. Also note that we did not subtract the mean vector as in the original expression. The reason is that subtracting the mean simply changes the origin of the transformed coordinates. We are interested in placing the outputs in a position similar to the inputs, and this is easier to do by extracting location information directly from the data. We do this as follows:



a b c
d e f

FIGURE 12.34 First row: Original characters. Second row: Characters aligned using principal components.

```
>> miny1 = min(Y(:, 1));
>> miny2 = min(Y(:, 2));
>> y1 = round(Y(:, 1) - miny1 + min(x1));
>> y2 = round(Y(:, 2) - miny2 + min(x2));
```

where the last two commands displace the coordinates so that the minimum coordinates will be approximately the same as for the original data before transformation.

The final step is to form an output image from the transformed (Y) data:

```
>> idx = sub2ind(size(f), y1, y2);
>> fout = false(size(f)); % Same size as input image.
>> fout(idx) = 1;
```

The first command forms a linear index from the transformed coordinates, and the last statement sets those coordinates to 1. The transformation from X to Y, and the rounding operation used in the formation of y1 and y2, generally create small gaps (0-valued pixels) in the region of the output objects. These are filled by dilating and then eroding (i.e., closing) the data with a 3×3 structuring element:

```
>> fout = imclose(fout, ones(3));
```

Finally, displaying this image would show that the letter A in the figure is upside down. In general, the principal components transform aligns the data along the direction of its principal spread, but there is no guarantee that the alignment will not be 180° in the opposite direction. To guarantee this would require that some "intelligence" be built into the process. That is beyond the present discussion, so we use visual analysis to rotate the data so that the letter is oriented properly.

```
>> fout = rot90(fout, 2);
>> imshow(fout) % Figure 12.34(d).
```

As the result in Fig. 12.34(d) shows, the method did a reasonable job of aligning the object along its principal direction. The coordinates in Fig. 12.34(a) are (x_1, x_2) while in Fig. 12.34(d) the coordinates are (y_1, y_2). An important characteristic of the approach just discussed is that it uses all the coordinate points of the input (contained in X) in forming the transformation matrix used to obtain the output. Hence, the method is reasonably insensitive to outliers. The results in Figs. 12.34(e) and (f) were generated in a similar manner. ■

Summary

The representation and description of objects or regions that have been segmented out of an image is an early step in the preparation of image data for subsequent use in automation. Descriptors such as the ones covered in this chapter constitute the input to

the object recognition algorithms developed in the next chapter. The custom functions developed in the preceding sections are a significant enhancement of the power of the Image Processing Toolbox functions available for image representation and description. It should be clear by now that the choice of one type of descriptor over another is dictated to a large degree by the problem at hand. This is one of the principal reasons why the solution of image processing problems is aided significantly by having a flexible prototyping environment in which existing functions can be integrated with new code to gain flexibility and reduce development time. The material in this chapter is a good example of how to construct the basis for such an environment.



13 Object Recognition

Preview

We conclude the book with a discussion and development of several M-functions for region and/or boundary recognition, which in this chapter we call *objects* or *patterns*. Approaches to computerized pattern recognition may be divided into two principal areas: decision-theoretic and structural. The first category deals with patterns described using quantitative descriptors, such as length, area, texture, and many of the other descriptors discussed in Chapter 12. The second category deals with patterns best represented by symbolic information, such as strings, and described by the properties and relationships between those symbols, as explained in Section 13.4. Central to the theme of recognition is the concept of "learning" from sample patterns. Learning techniques for both decision-theoretic and structural approaches are discussed in the material that follows.

13.1 Background

A *pattern* is an arrangement of descriptors, such as those discussed in Chapter 12. The name *feature* is used interchangeably in the pattern recognition literature to denote a descriptor. A *pattern class* is a family of patterns that share a set of common properties. Pattern classes are denoted $\omega_1, \omega_2, \ldots, \omega_W$ where W is the number of classes. Pattern recognition by machine involves techniques for assigning patterns to their respective classes—automatically and with as little human intervention as possible.

The two principal pattern arrangements used in practice are vectors (for quantitative descriptions) and strings (for structural descriptions). Pattern vectors are represented by bold lowercase letters, such as $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, and have the $n \times 1$ vector form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

where component x_i represents the ith descriptor and n is the total number of such descriptors associated with the pattern. Sometimes, it is necessary in computations to use row vectors of dimension $1 \times n$, which are obtained by forming the transpose, $\mathbf{x}^T$, of the preceding column vector.

The nature of the components of a pattern vector $\mathbf{x}$ depends on the approach used to describe the physical pattern itself. For example, consider the problem of automatically classifying alphanumeric characters. Descriptors suitable for a decision-theoretic approach might include measures such as 2-D moment invariants or a set of Fourier coefficients describing the outer boundary of the characters.

In some applications, pattern characteristics are best described by structural relationships. For example, fingerprint recognition is based on the interrelationships of print features called *minutiae*. Together with their relative sizes and locations, these features are primitive components that describe fingerprint ridge properties, such as abrupt endings, branching, merging, and disconnected segments. Recognition problems of this type, in which quantitative measures about each feature, and the spatial relationships between the features, determine class membership, generally are best solved by structural approaches.

The material in the following sections is representative of techniques for implementing pattern recognition solutions in MATLAB. A basic concept in recognition, especially in decision-theoretic applications, is the idea of pattern matching based on measures of distance between pattern vectors. Therefore, we begin our discussion with various approaches for the efficient computation of distance measures in MATLAB.

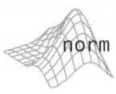
13.2 Computing Distance Measures in MATLAB

The material in this section deals with vectorizing distance computations that otherwise would involve `for` or `while` loops. Some of the vectorized expressions are more subtle than most of the vectorized code in previous chapters, so you are encouraged to study them in detail. The following formulations are based on a summary of similar expressions compiled by Acklam [2002].

The *Euclidean distance* between two n-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as the *scalar*

$$D(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| = \|\mathbf{y} - \mathbf{x}\| = \left[(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2\right]^{\frac{1}{2}}$$

This expression is the norm of the difference between the two vectors, so we compute it using MATLAB's function `norm`:




```
D = norm(x - y)
```

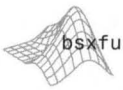
where x and y are vectors corresponding to $\mathbf{x}$ and $\mathbf{y}$ in the preceding equation for $D(\mathbf{x}, \mathbf{y})$.

Often, it is necessary to compute a *set* of Euclidean distances between a vector $\mathbf{y}$ and each vector of a vector *population* consisting of p, n-dimensional vectors arranged as the *rows* of a $p \times n$ matrix $\mathbf{X}$. For the dimensions to line up properly, $\mathbf{y}$ has to be of dimension $1 \times n$. Then, the distance between $\mathbf{y}$ and each row of $\mathbf{X}$ is contained in the $p \times 1$ vector

```
D = sqrt(sum(abs(X - repmat(y, p, 1)).^2, 2));
```

where `D(i)` is the Euclidean distance between `y` and the ith row of `X` [i.e., `X(i, :)`]. Note the use of function `repmat` to duplicate row vector `y` p times and thus form a $p \times n$ matrix to match the dimensions of `X`. The last 2 on the right of the preceding line of code indicates that `sum` is to operate along dimension 2; that is, to sum the elements along the horizontal dimension.

Although the preceding `repmat` formulation makes explicit the need to match matrix dimensions, a newer MATLAB function, `bsxfun`, performs the same operation using less memory, and (usually) it runs faster. The syntax is




```
C = bsxfun(fun, A, B)
```

Many of the formulations given by Acklam [2002] (see the first paragraph in this section) use function `repmat`. Function `bsxfun` just provides a more efficient implementation of his original expressions.

This function applies an element by element operation to arrays `A` and `B`, as defined by `fun`, which is a function handle that can be either one of the built-in functions in Table 13.1, or a user-defined M-file function. For example, suppose that

```
X =
     1     2
     3     4
     5     6
```

and

TABLE 13.1 Built-in functions for function `bsxfun`.

Function	Explanation	Function	Explanation	Function	Explanation
`@plus`	Plus	`@min`	Minimum	`@lt`	Less than
`@minus`	Minus	`@rem`	Remainder after division	`@le`	Less than or equal to
`@times`	Array multiply	`@mod`	Modulus after division	`@gt`	Greater than
`@rdivide`	Right array divide	`@atan2`	4-quadrant arctangent	`@ge`	Greater than or equal to
`@ldivide`	Left array divide	`@hypot`	Sq. root of sum of squares	`@and`	Logical AND
`@power`	Array power	`@eq`	Equal	`@or`	Logical OR
`@max`	Maximum	`@ne`	Not equal	`@xor`	Logical exclusive OR

```
y =
    1    3
```

Then

```
>> bsxfun(@minus, X, y)
ans =
    0   -1
    2    1
    4    3
```

Note that `bsxfun` expanded the singleton dimension of `y` (the number of rows) to match the dimensions of `X`. Of course, the operations specified must be meaningful. For example, if `y` had been a column vector instead, subtracting `y` from `X` would be meaningless, and `bsxfun` would issue the error: "Non-singleton dimensions of the two input arrays must match each other."

Recall from Section 2.2 that a singleton dimension is any dimension `dim` for which `size(A, dim) = 1`.

Using `bsxfun`, the preceding distance equation becomes

```
D = sqrt(sum(abs(bsxfun(@minus, X, y)).^2, 2));
```

As you can see, this is a more compact and clearer form.

Suppose next that we have two vector populations $\mathbf{X}$, of dimension $p \times n$ and $\mathbf{Y}$ of dimension $q \times n$. The matrix containing the distances between rows of these two populations can be obtained using the expression

```
D = sqrt(sum(abs(bsxfun(@minus, permute(X, [1 3 2]),...
                permute(Y,[3 1 2]))).^2, 3));
```

where `D` is now a *matrix* of size $p \times q$, whose element `D(i, j)` is the Euclidean distance between the ith and jth rows of the populations; that is, the distance between `X(i, :)` and `Y(j, :)`.

The syntax for function `permute` in the preceding expression is

```
B = permute(A, order)
```



This function reorders the dimensions of `A` according to the elements of the vector `order` (the elements of this vector must be unique). For example, if `A` is a 2-D array, the statement `B = permute(A, [2 1])` interchanges the rows and columns of `A`, which is equivalent to letting `B` equal the transpose of `A`. If the length of vector `order` is greater than the number of dimensions of `A`, MATLAB processes the components of the vector from left to right, until all elements are used. In the preceding expression for `D`, `permute(X, [1 3 2])` creates arrays in the third dimension, each being a column (dimension 1) of `X`. Because there are n columns in `X`, n such arrays are created, with each array being of dimension $p \times 1$. Therefore, the command `permute(X, [1 3 2])` creates an array of dimension $p \times 1 \times n$.

Recall from Section 2.2 that the first dimension of a matrix `A` is along the vertical (row locations) and the second along the horizontal (column locations). Thus, swapping the dimensions of `A` is the same as transposing the matrix.

Similarly, the command `permute(Y, [3 1 2])` creates an array of dimension $1 \times q \times n$. Fundamentally, the preceding expressions for `D` are vectorizations of the expressions that could be written using `for` or `while` loops.

In addition to the expressions just discussed, we use in this chapter a distance measure from a vector $\mathbf{y}$ to the mean $\mathbf{m}_x$ of a vector population, weighted inversely by the covariance matrix, $\mathbf{C}_x$, of the population. This metric, called the *Mahalanobis distance*, is defined as

$$D(\mathbf{y}, \mathbf{m}_x) = (\mathbf{y} - \mathbf{m}_x)^T \mathbf{C}_x^{-1} (\mathbf{y} - \mathbf{m}_x)$$

The inverse matrix operation is the most time-consuming computational task required to implement the Mahalanobis distance. This operation can be optimized significantly by using MATLAB's matrix right division operator (/) introduced in Table 2.5 (see also the margin note in the following page). Expressions for $\mathbf{m}_x$ and $\mathbf{C}_x$ are given in Section 12.5.

Let $\mathbf{X}$ denote a population of p, n-dimensional vectors, and let $\mathbf{Y}$ denote a population of q, n-dimensional vectors, such that the vectors in both $\mathbf{X}$ and $\mathbf{Y}$ are the rows of these arrays. The objective of the following M-function is to compute the Mahalanobis distance between every vector in $\mathbf{Y}$ and the mean, $\mathbf{m}_x$.

mahalanobis

```
function D = mahalanobis(varargin)
%MAHALANOBIS Computes the Mahalanobis distance.
%   D = MAHALANOBIS(Y, X) computes the Mahalanobis distance between
%   each vector in Y to the mean (centroid) of the vectors in X, and
%   outputs the result in vector D, whose length is size(Y, 1). The
%   vectors in X and Y are assumed to be organized as rows. The
%   input data can be real or complex. The outputs are real
%   quantities.
%
%   D = MAHALANOBIS(Y, CX, MX) computes the Mahalanobis distance
%   between each vector in Y and the given mean vector, MX. The
%   results are output in vector D, whose length is size(Y, 1). The
%   vectors in Y are assumed to be organized as the rows of this
%   array. The input data can be real or complex. The outputs are
%   real quantities. In addition to the mean vector MX, the
%   covariance matrix CX of a population of vectors X must be
%   provided also. Use function COVMATRIX (Section 12.5) to compute
%   MX and CX.

% Reference: Acklam, P. J. [2002]. "MATLAB Array Manipulation Tips
% and Tricks," available at
%     home.online.no/~pjacklam/matlab/doc/mtt/index.html
% or in the Tutorials section at
%     www.imageprocessingplace.com

param = varargin; % Keep in mind that param is a cell array.
Y = param{1};
```

```
if length(param) == 2
   X = param{2};
   % Compute the mean vector and covariance matrix of the vectors
   % in X.
   [Cx, mx] = covmatrix(X);
elseif length(param) == 3 % Cov. matrix and mean vector provided.
   Cx = param{2};
   mx = param{3};
else
   error('Wrong number of inputs.')
end
mx = mx(:)'; % Make sure that mx is a row vector for the next step.

% Subtract the mean vector from each vector in Y.
Yc = bsxfun(@minus, Y, mx);

% Compute the Mahalanobis distances.
D = real(sum(Yc/Cx.*conj(Yc), 2));
```

The MATLAB matrix operation `A/B` is more accurate (and generally faster) than the operation `B*inv(A)`. Similarly, `A\B` is preferred to `inv(A)*B`. It is assumed that the sizes of `A` and `B` are compatible for these operations to be defined. See Table 2.5.

The call to `real` in the last line of code is to remove "numeric noise" if earlier versions of MATLAB are used. If the data are known to always be real, the code can be simplified by removing functions `real` and `conj`.

13.3 Recognition Based on Decision-Theoretic Methods

Decision-theoretic approaches to recognition are based on the use of *decision* (also called *discriminant*) *functions*. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ denote an n-dimensional pattern vector, as discussed in Section 13.1. For W pattern classes, $\omega_1, \omega_2, \ldots, \omega_W$, the basic problem in decision-theoretic pattern recognition is to find W decision functions, $d_1(\mathbf{x}), d_2(\mathbf{x}), \ldots, d_W(\mathbf{x})$, with the property that, if a pattern $\mathbf{x}$ belongs to class ω_i, then

$$d_i(\mathbf{x}) > d_j(\mathbf{x}) \quad j = 1, 2, \ldots, W; \quad j \neq i$$

In other words, an unknown pattern $\mathbf{x}$ is said to belong to the ith pattern class if, upon substitution of $\mathbf{x}$ into all decision functions, $d_i(\mathbf{x})$ yields the largest numerical value. Ties are resolved arbitrarily.

The *decision boundary* separating class ω_i from ω_j is given by values of $\mathbf{x}$ for which $d_i(\mathbf{x}) = d_j(\mathbf{x})$ or, equivalently, by values of $\mathbf{x}$ for which

$$d_i(\mathbf{x}) - d_j(\mathbf{x}) = 0$$

Common practice is to express the decision boundary between two classes by the single function $d_{ij}(\mathbf{x}) = d_i(\mathbf{x}) - d_j(\mathbf{x})$. Thus $d_{ij}(\mathbf{x}) > 0$ for patterns of class ω_i and $d_{ij}(\mathbf{x}) < 0$ for patterns of class ω_j. If $d_i(\mathbf{x}) = d_j(\mathbf{x})$, then pattern $\mathbf{x}$ lies on the boundary between the two classes.

As will become clear in the following sections, finding decision functions entails estimating parameters from patterns that are representative of the classes of interest. Patterns used for parameter estimation are called *training patterns*, or *training sets*. Sets of patterns of known classes that are not used for training, but are used instead to test the performance of a particular recognition approach are referred to as *test* or *independent patterns* or *sets*. The principal objective of Sections 13.3.2 and 13.3.4 is to develop various approaches for finding decision functions based on parameter estimation using training sets.

13.3.1 Forming Pattern Vectors

As noted at the beginning of this chapter, pattern vectors can be formed from quantitative descriptors, such as those discussed in Chapter 12 for regions and/or boundaries. For example, suppose that we describe a boundary by using Fourier descriptors. The value of the ith descriptor becomes the value of x_i, the ith component of a pattern vector. In addition, we could append other components to pattern vectors. For instance, we could incorporate six additional components to the Fourier-descriptor by appending to each vector the six measures of texture in Table 12.2.

An approach used when dealing with registered multispectral images is to stack the images and then form vectors from corresponding pixels in the images, as illustrated in Fig. 12.29. The images are stacked using function `cat`:

```
S = cat(3, f1, f2, ..., fn)
```

where `S` is the stack and `f1, f2, ..., fn` are the images from which the stack is formed. The vectors then are generated by using function `imstack2vectors` discussed in Section 12.5. See Example 13.2 for an illustration.

13.3.2 Pattern Matching Using Minimum-Distance Classifiers

Suppose that each pattern class, ω_j, is characterized by a mean vector $\mathbf{m}_j$. That is, we use the mean vector of each population of training vectors as being representative of that class of vectors:

$$\mathbf{m}_j = \frac{1}{N_j} \sum_{\mathbf{x} \in \omega_j} \mathbf{x}$$

where N_j is the number of training pattern vectors from class ω_j and the summation is taken over these vectors. As before, W is the number of pattern classes. One way to determine the class membership of an *unknown* pattern vector $\mathbf{x}$ is to assign it to the class of its closest prototype. Using the Euclidean distance as a measure of closeness (i.e., similarity) reduces the problem to computing the distance measures:

$$D_j(\mathbf{x}) = \|\mathbf{x} - \mathbf{m}_j\| \qquad j = 1, 2, \ldots, W$$

We then assign $\mathbf{x}$ to class ω_i if $D_i(\mathbf{x})$ is the smallest distance. That is, the smallest distance implies the best match in this formulation.

Suppose that all the mean vectors are organized as rows of a matrix **M**. Then, computing the distances from an arbitrary pattern **x** to all the mean vectors is accomplished by using the expression discussed in Section 13.2:

```
D = sqrt(sum(abs(bsxfun(@minus, N, x)).^2, 2))
```

Because all distances are positive, this statement can be simplified by ignoring the `sqrt` operation.

The minimum of `D` determines the class membership of pattern vector `x`:

```
>> xclass = find(D == min(D));
```

If more than one minimum exists, `xclass` would equal a vector, with each of its elements pointing to a different pattern class. In this case, the class membership cannot be determined uniquely.

If, instead of a single pattern, we have a set of patterns arranged as the rows of a matrix, `X`, then we use an expression similar to the longer expression in Section 13.2 to obtain a *matrix* `D`, whose element `D(i, j)` is the Euclidean distance between the ith pattern vector in `X` and the jth mean vector in `M`. Thus, to find the class membership of, say, the ith pattern in `X`, we find the column location in row `i` of `D` that yields the smallest value. Multiple minima yield multiple values, as in the single-vector case discussed in the last paragraph.

In order to reduce proliferation of notation, we use D and `D` to denote both a scalar distance and a matrix of distances. Lowercase d and `d` are used to denote decision functions.

It is not difficult to show that selecting the smallest distance is equivalent to evaluating the functions

$$d_j(\mathbf{x}) = \mathbf{x}^T\mathbf{m}_j - \frac{1}{2}\mathbf{m}_j^T\mathbf{m}_j \quad j = 1, 2, \ldots, W$$

and assigning **x** to class ω_i if $d_i(\mathbf{x})$ yields the *largest* numerical value. This formulation agrees with the concept of a decision function defined earlier.

The decision boundary between classes ω_i and ω_j for a minimum distance classifier is

$$\begin{aligned} d_{ij}(\mathbf{x}) &= d_i(\mathbf{x}) - d_j(\mathbf{x}) \\ &= \mathbf{x}^T(\mathbf{m}_i - \mathbf{m}_j) - \frac{1}{2}(\mathbf{m}_i - \mathbf{m}_j)^T(\mathbf{m}_i + \mathbf{m}_j) = 0 \end{aligned}$$

The surface defined by this equation is the perpendicular bisector of the line segment joining $\mathbf{m}_i$ and $\mathbf{m}_j$. For $n = 2$ the perpendicular bisector is a line, for $n = 3$ it is a plane, and for $n > 3$ it is called a *hyperplane*.

13.3.3 Matching by Correlation

Given an image $f(x, y)$, the correlation problem is to find all places in the image that match a given subimage $w(x, y)$ (called a *mask* or *template*). Usually, $w(x, y)$ is much smaller than $f(x, y)$. The method of choice for

A more formal term for the correlation of two different functions is *cross-correlation*. When the functions are the same, correlation is referred to as *autocorrelation*. Often, when the meaning is clear, the generic term *correlation* is used to denote either auto-, or cross-correlation, as we do here.

matching by correlation is to use the correlation coefficient, which we know from Chapter 6 is defined as

$$\gamma(x,y) = \frac{\sum_{s,t}\left[w(s,t) - \bar{w}\right]\left[f(x+s,y+t) - \bar{f}_{xy}\right]}{\sqrt{\sum_{s,t}\left[w(s,t) - \bar{w}\right]^2 \sum_{s,t}\left[f(x+s,y+t) - \bar{f}_{xy}\right]^2}}$$

where w is the template, $\bar{w}$ is the average value of the elements of the template (computed only once), f is the image, and $\bar{f}_{xy}$ is the average value of the image in the region where f and w overlap. The summation is taken over the values of s and t such that the image and the template overlap. The denominator normalizes the result with respect to variations in intensity. The values of $\gamma(x,y)$ are in the range $[-1, 1]$. A high value of $|\gamma(x,y)|$ generally indicates a good match[†] between the template and the image, when the template is centered at coordinates (x,y). As noted in Section 6.7.5, the correlation coefficient is computed by toolbox function `normxcorr2`:

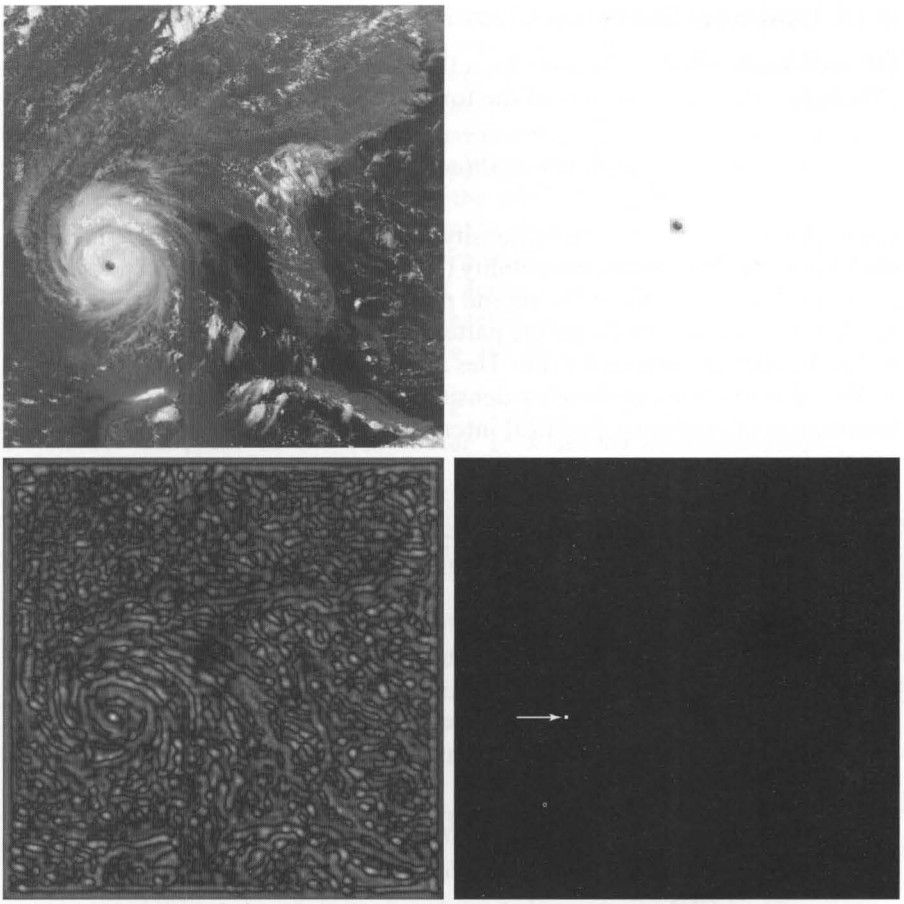
```
g = normxcorr2(template, f)
```

EXAMPLE 13.1: Using correlation for image matching.

■ Figure 13.1(a) shows an image of Hurricane Andrew, in which the eye of the storm is clearly visible. As an example of correlation, we wish to find the location of the best match in Fig. 13.1(a) of the eye subimage (i.e., the template) in Fig. 13.1(b). The sizes of the image and template are 912×912 and 32×32 pixels, respectively. Figure 13.1(c) is the result of the following commands:

```
>> f = imread('Fig1301(a).tif');
>> w = imread('Fig1301(b).tif');
>> g = abs(normxcorr2(w, f));
>> imshow(g) % Fig. 13.1(c)
>> % Find all the max values.
>> gT = g == max(g(:)); % gT is a logical array.
>> % Find out how many peaks there are.
>> idx = find(gT == 1); % We use idx again later.
>> numel(idx)
ans =
    1
>> % A single point is hard to see. Increase its size.
>> gT = imdilate(gT, ones(7));
>> figure, imshow(gT) % Fig. 13.1(d).
```

[†] Terms such as "high" and "good" are relative when referring to correlation. For example, in the case of a low resolution imaging sensor operating in an unconstrained environment, a correlation value of, say, 0.9 might indicate a good, acceptable match. On the other hand, when referring to a very high quality imaging sensor in a controlled environment, the same value of correlation might be well below what is considered a good match.



a b
c d

FIGURE 13.1
(a) Image of Hurricane Andrew. (b) Template. (c) Correlation of image and template. (d) Location of the best match. (The single point marking the best match was enlarged to make it easier to see). (Original image courtesy of NOAA.)

The blurring evident in the correlation image of Fig. 13.1(c) should not be a surprise because the template in 13.1(b) has two dominant, nearly constant regions, and thus behaves similarly to a lowpass filter. The brightest area in Fig. 13.1(c) corresponds to the best match between the template and the original image. As you can see, the best match corresponds quite closely with the location of the eye of the storm in Fig. 13.1(a).

In general, the feature of interest is the location of the best match (or matches) which, for correlation, implies finding the location(s) of the highest value in the correlation image. We find the location(s) of the peak(s) as follows:

```
>> [r, c] = ind2sub(size(f), idx);
[r c]
ans =
   605    246
```

which, in this case, is only one peak, as Fig. 13.1(d) shows. ■

13.3.4 Optimum Statistical Classifiers

The well-known Bayes classifier for a 0-1 loss function (Gonzalez and Woods [2008]) has decision functions of the form

$$d_j(\mathbf{x}) = p(\mathbf{x}/\omega_j)P(\omega_j) \quad j = 1, 2, \ldots, W$$

where $p(\mathbf{x}/\omega_j)$ is the probability density function (PDF) of the pattern vectors of class ω_j and $P(\omega_j)$ is the probability (a scalar) that class ω_j occurs. As before, given an unknown pattern vector, the process is to compute a total of W decision functions and then assign the pattern to the class whose decision function yields the largest numerical value. Ties are resolved arbitrarily.

The case when the probability density functions are (or are assumed to be) Gaussian is of particular practical interest. The n-dimensional Gaussian PDF has the form

$$p(\mathbf{x}/\omega_j) = \frac{1}{(2\pi)^{n/2}\,|\mathbf{C}_j|^{1/2}}\, e^{-\frac{1}{2}(\mathbf{x}-\mathbf{m}_j)^T\mathbf{C}_j^{-1}(\mathbf{x}-\mathbf{m}_j)}$$

where $\mathbf{C}_j$ and $\mathbf{m}_j$ are the covariance matrix and mean vector of the pattern population of class ω_j and $|\mathbf{C}_j|$ is the determinant of $\mathbf{C}_j$.

Because the logarithm is a monotonically increasing function, choosing the largest $d_j(\mathbf{x})$ to classify patterns is equivalent to choosing the largest $\ln\left[d_j(\mathbf{x})\right]$ so we can use instead decision functions of the form

$$\begin{aligned} d_j(\mathbf{x}) &= \ln\left[p(\mathbf{x}/\omega_j)P(\omega_j)\right] \\ &= \ln p(\mathbf{x}/\omega_j) + \ln P(\omega_j) \end{aligned}$$

where the logarithm is guaranteed to be real because $p(\mathbf{x}/\omega_j)$ and $P(\omega_j)$ are nonnegative. Substituting the expression for the Gaussian PDF gives the equation

$$d_j(\mathbf{x}) = \ln P(\omega_j) - \frac{n}{2}\ln 2\pi - \frac{1}{2}\ln|\mathbf{C}_j| - \frac{1}{2}\left[(\mathbf{x}-\mathbf{m}_j)^T\mathbf{C}_j^{-1}(\mathbf{x}-\mathbf{m}_j)\right]$$

The term $(n/2)\ln 2\pi$ is a positive constant that is independent of the class of any pattern, so it can be deleted, yielding the decision functions

$$d_j(\mathbf{x}) = \ln P(\omega_j) - \frac{1}{2}\ln|\mathbf{C}_j| - \frac{1}{2}\left[(\mathbf{x}-\mathbf{m}_j)^T\mathbf{C}_j^{-1}(\mathbf{x}-\mathbf{m}_j)\right]$$

for $j = 1, 2, \ldots, W$. The term inside the brackets is recognized as the Mahalanobis distance discussed in Section 13.2, for which we have a vectorized implementation. We also have an efficient method for computing the mean and covariance matrix from Section 12.5, so implementing the Bayes classifier for the multivariate Gaussian case is straightforward, as the following function shows.

bayesgauss

```
function d = bayesgauss(X, CA, MA, P)
%BAYESGAUSS Bayes classifier for Gaussian patterns.
%   D = BAYESGAUSS(X, CA, MA, P) computes the Bayes decision
%   functions of the n-dimensional patterns in the rows of X. CA is
%   an array of size n-by-n-by-W containing W covariance matrices of
%   size n-by-n, where W is the number of classes. MA is an array of
%   size n-by-W, whose columns are the corresponding mean vectors. A
%   covariance matrix and a mean vector must be specified for each
%   class. X is of size K-by-n, where K is the number of patterns
%   to be classified. P is a 1-by-W array, containing the
%   probabilities of occurrence of each class. If P is not included
%   in the argument, the classes are assumed to be equally likely.
%
%   D is a column vector of length K. Its ith element is the class
%   number assigned to the ith vector in X during classification.

% Verify number of inputs.
error(nargchk(3, 4, nargin))
n = size(CA, 1); % Dimension of patterns.

% Protect against the possibility that the class number is included
% as an (n + 1)th element of the vectors.
X = double(X(:, 1:n));
W = size(CA, 3); % Number of pattern classes.
K = size(X, 1);  % Number of patterns to classify.
if nargin == 3
   P(1:W) = 1/W; % Classes assumed equally likely.
else
   if sum(P) ~= 1
      error('Elements of P must sum to 1.');
   end
end
% Compute the determinants.
for J = 1:W
   DM(J) = det(CA(:, :, J));
end

% Evaluate the decision functions. Note the use of function
% mahalanobis discussed in Section 13.2.
MA = MA'; % Organize the mean vectors as rows.
for J = 1:W
   C = CA(:, :, J);
   M = MA(J, :);
   L(1:K, 1) = log(P(J));
   DET(1:K, 1) = 0.5*log(DM(J));
   if P(J) == 0;
      D(1:K, J) = -inf;
   else
      D(:, J) = L - DET - 0.5*mahalanobis(X, C, M);
```

```
    end
end

% Find the coordinates of the maximum value in each row. These
% maxima give the class of each pattern.
[i, j] = find(bsxfun(@eq, D, max(D, [], 2)));
% Re-use X. It contains now the max value along each column.
X = [i j];
% Eliminate multiple classifications of the same patterns. Since
% the class assignment when two or more decision functions give
% the same value is arbitrary, we need to keep only one.
X = sortrows(X);
[b, m] = unique(X(:, 1));
X = X(m, :);
% X is now sorted, with the 2nd column giving the class of the
% pattern number in the 1st col.;  i.e., X(j, 1) refers to the jth
% input pattern, and X(j, 2) is its class number.

% Output the result of classification. d is a column vector with
% length equal to the total number of input patterns. The elements
% of d are the classes into which the patterns were classified.
d = X(:, 2);
```

`max(D, [ ], 2)` finds the maximum of `D` along its second dimension (its rows). The result is a vector of size `size(D, 1)-by-1`.

EXAMPLE 13.2: Bayes classification of multispectral data.

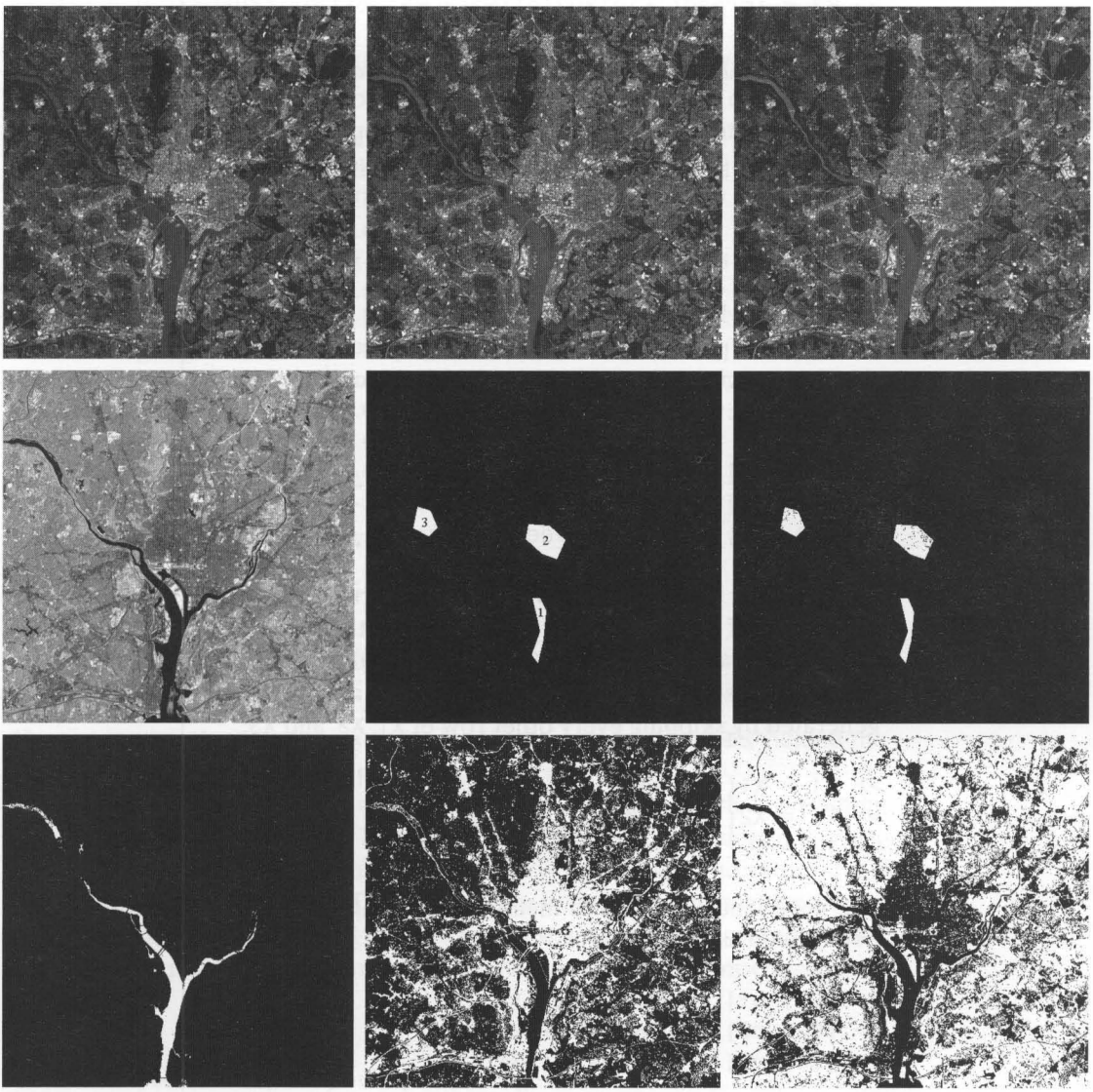
■ Bayes recognition is used frequently to automate the classification of regions in multispectral imagery. Figure 13.2 shows the first four images from Fig. 12.30 (three visual bands and one infrared band). The objective of this example is to use the Bayes classifier to classify the pixels in these images into three classes: *water*, *urban*, and *vegetation*. The pattern vectors in this example are formed by the method discussed in Sections 12.5 and 13.3.1, in which corresponding pixels in the images are organized as vectors. We are dealing with four images, so the pattern vectors are four dimensional. The images were read using the statements:

```
>> f1 = imread('Fig1302(a)(WashingtonDC_Band1_512).tif');
>> f2 = imread('Fig1302(b)(WashingtonDC_Band2_512).tif');
>> f3 = imread('Fig1302(c)(WashingtonDC_Band3_512).tif');
>> f4 = imread('Fig1302(d)(WashingtonDC_Band4_512).tif');
```

To obtain the mean vectors and covariance matrices, we need samples representative of each pattern class. A simple way to obtain such samples interactively is to use function `roipoly` (see Section 5.2.4) with the statement

```
>> B = roipoly(f);
```

where `f` is any of the multispectral images and `B` is a binary mask image. With this format, image `B` is generated interactively on the screen. Figure 13.2(e) shows three mask images, `B1`, `B2`, and `B3`, generated using this method. The numbers 1, 2, and 3 identify regions containing samples representative of water, urban development, and vegetation, respectively. The images were saved to disk and then read using the statements



a b c
d e f
g h i

FIGURE 13.2 Bayes classification of multispectral data. (a)-(d) Images in the visible blue, visible green, visible red, and near infrared wavelengths. (e) Masks showing sample regions of (1) water, (2) urban development, and (3) vegetation. (f) Results of classification; the black dots denote points classified incorrectly. The other (white) points were classified correctly. (g) All image pixels classified as water (in white). (h) All image pixels classified as urban development (in white). (i) All image pixels classified as vegetations (in white). All images are of size 512×512 pixels.

```
>> B1 = imread('Fig1302(e)(Mask_B1).tif');
>> B2 = imread('Fig1302(e)(Mask_B2).tif');
>> B3 = imread('Fig1302(e)(Mask_B3).tif');
```

Figure 13.2(e) was generated by ORing these masks, B1 | B2 | B3, (the numbers in the figure are for explanation only; they are not part of the data).

The next step is to obtain the vectors corresponding to each region. The four images are registered spatially, so they simply are concatenated along the third dimension to obtain an image stack, as in Section 12.5:

```
>> stack = cat(3, f1, f2, f3, f4);
```

Any point, when viewed through these four images, corresponds to a four-dimensional pattern vector. We are interested in the vectors contained in the three regions shown in Fig. 13.2(e), which we obtain by using function imstack2vectors discussed in Section 12.5:

```
>> [X1, R1] = imstack2vectors(stack, B1);
>> [X2, R2] = imstack2vectors(stack, B2);
>> [X3, R3] = imstack2vectors(stack, B3);
```

where X is an array whose rows are the pattern vectors, and R contains the linear indices of the location of those vectors in the region defined by B.

Three subsets, T1, T2, and T3 were extracted from the X's for use as training samples to estimate the covariance matrices and mean vectors. The T's were generated by skipping every other row of X1, X2, and X3:

```
>> T1 = X1(1:2:end, :);
>> T2 = X2(1:2:end, :);
>> T3 = X3(1:2:end, :);
```

The covariance matrix and mean vector of each training data set were then determined as follows:

```
>> [C1, m1] = covmatrix(T1);
>> [C2, m2] = covmatrix(T2);
>> [C3, m3] = covmatrix(T3);
```

Then, we formed arrays CA and MA for use in function bayesgauss, as follows:

```
>> CA = cat(3, C1, C2, C3);
>> MA = cat(2, m1, m2, m3);
```

The performance of the classifier with the training patterns was determined by classifying the training sets, where we assumed that all $P(\omega_i)$ were equal (i.e., the classes were equally likely to occur):

```
>> dT1 = bayesgauss(T1, CA, MA);
>> dT2 = bayesgauss(T2, CA, MA);
>> dT3 = bayesgauss(T3, CA, MA);
```

The results of classifying the training data were tabulated as follows:

```
>> % Number of training patterns class_k_to_class1, k = 1, 2, 3.
>> class1_to_1 = numel(find(dT1==1));
>> class1_to_2 = numel(find(dT1==2));
>> class1_to_3 = numel(find(dT1==3));
>> % Number of training patterns class_k_to_class2, k = 1, 2, 3.
>> class2_to_1 = numel(find(dT2==1));
>> class2_to_2 = numel(find(dT2==2));
>> class2_to_3 = numel(find(dT2==3));
>> % Number of training patterns class_k_to_class3, k = 1, 2, 3.
>> class3_to_1 = numel(find(dT3==1));
>> class3_to_2 = numel(find(dT3==2));
>> class3_to_3 = numel(find(dT3==3));
```

The independent pattern sets were formed as

```
>> I1 = X1(2:2:end, :);
>> I2 = X2(2:2:end, :);
>> I3 = X3(2:2:end, :);
```

Then, repeating the preceding steps using the `I`'s instead of the `T`'s yielded the recognition results for the independent pattern set.

Table 13.2 summarizes the recognition results obtained with the training and independent pattern sets. The percentage of training and independent patterns recognized correctly was about the same with both sets, indicating stability in the parameter estimates. The largest error in both cases was with patterns from the urban area. This is not unexpected, as vegetation is present there also (note that no patterns in the urban or vegetation areas were misclassified as water).

TABLE 13.2 Bayes classification of multispectral image data.

Training Patterns						Independent Patterns					
		Classified into Class						Classified into Class			
Class	No. of Samples	1	2	3	% Correct	Class	No. of Samples	1	2	3	% Correct
1	484	482	2	0	99.6	1	483	478	3	2	98.9
2	933	0	885	48	94.9	2	932	0	880	52	94.4
3	483	0	19	464	96.1	3	482	0	16	466	96.7

Figure 13.2(f) shows as black dots the points that were misclassified and as white dots the points that were classified correctly in each region (for all patterns in the training and independent sets). No black dots are readily visible in region 1 because the 7 misclassified points are very close to, or on, the boundary of the white region. To generate, for example, the classification results in region B2, we used the following commands:

```
>> image2 = false(size(f2));
>> d2 = bayesgauss(X2, CA, MA);
>> idx2 = find(d2 == 2);
>> image2(R2(idx2)) = 1;
```

and similarly for the other two regions. A composite image was then generated for display:

```
>> compositeImage = image1 | image2 | image3; % Fig. 13.2(f).
```

Figures 13.2(g) through (i) are more interesting. Here, we used the mean vectors and covariance matrices obtained from the training data to classify *all* image pixels into one of the three categories, using the commands:

```
>> B = ones(size(f1)); % This B selects all patterns.
>> X = imstack2vectors(stack, B);
>> dAll = bayesgauss(X, CA, MA); % Classify all patterns.
>> image_class1 = reshape(dAll == 1, 512, 512);
>> image_class2 = reshape(dAll == 2, 512, 512);
>> image_class3 = reshape(dAll == 3, 512, 512);
>> figure, imshow(image_class1) % Fig. 13.2(g).
>> figure, imshow(image_class2) % Fig. 13.2(h).
>> figure, imshow(image_class3) % Fig. 13.2(i).
```

Note that R's were not used in function `imstack2vectors` because B encompasses the entire image area.

Figure 13.2(g) shows in white (i.e., 1) all the pixels that were classified as water. Pixels not classified as water are shown in black. We see that the Bayes classifier did an excellent job of determining which parts of the image were water. Figure 13.2(h) shows in white all pixels classified as urban development; observe how well the system performed in recognizing urban features, such as the bridges and highways. Figure 13.2(i) shows the pixels classified as vegetation. The center area in Fig. 13.2(h) shows a high concentration of white pixels in the downtown area, with the density decreasing as a function of distance from the center of the image. Figure 13.2(i) shows the opposite effect, indicating the least vegetation toward the center of the image, where urban development is greatest. ■

13.3.5 Adaptive Learning Systems

The approaches discussed in Sections 13.3.1 and 13.3.3 are based on the use of sample patterns to estimate the statistical parameters of each pattern class. The minimum-distance classifier is specified completely by the mean vector of each class. Similarly, the Bayes classifier for Gaussian populations is specified completely by the mean vector and covariance matrix of each class of patterns.

In these two approaches, training is a simple matter. The training patterns of each class are used to compute the parameters of the decision function corresponding to that class. After the parameters in question have been estimated, the structure of the classifier is fixed, and its eventual performance will depend on how well the actual pattern populations satisfy the underlying statistical assumptions made in the derivation of the classification method being used.

The methods just discussed can be quite effective, provided that the pattern classes are characterized, at least approximately, by Gaussian probability density functions. When this assumption is not valid, designing a statistical classifier becomes a much more difficult task because estimating multivariate probability density functions is not a trivial endeavor. In practice, such decision-theoretic problems are best handled by methods that yield the required decision functions directly via training. Then, having to make assumptions regarding the underlying probability density functions or other probabilistic information about the pattern classes under consideration is not necessary.

The principal approach in use today for this type of classification is based on neural networks (Gonzalez and Woods [2008]). The scope of implementing neural networks suitable for image-processing applications is not beyond the capabilities of the functions available to us in MATLAB and the Image Processing Toolbox. However, this effort would be unwarranted in the present context because a comprehensive neural-networks toolbox has been available from The MathWorks for several years.

13.4 Structural Recognition

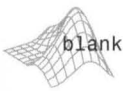
Structural recognition techniques are based generally on representing objects of interest as strings, trees, or graphs, and then defining descriptors and recognition rules based on those representations. The key difference between decision-theoretic and structural methods is that the former uses quantitative descriptors expressed in the form of numeric vectors. Structural techniques, on the other hand, deal principally with symbolic information. For instance, suppose that object boundaries in a given application are represented by minimum-perimeter polygons. A decision-theoretic approach might be based on forming vectors whose elements are the numeric values of the interior angles of the polygons, while a structural approach might be based on defining symbols for *ranges* of angle values and then forming a string of such symbols to describe the patterns.

Strings are by far the most common representation used in structural recognition, so we focus on this approach in this section. As will become evident shortly, MATLAB has an extensive set of specialized functions for string manipulation.

13.4.1 Working with Strings in MATLAB

In MATLAB, a string is a one-dimensional array whose components are the numeric codes for the characters in the string. The characters displayed depend on the character set used in encoding a given font. The *length* of a string is the number of characters in the string, including spaces. It is obtained using the familiar function `length`. A string is defined by enclosing its characters in single quotes (a textual quote within a string is indicated by two quotes).

Table 13.3 lists the principal MATLAB functions that deal with strings.[†] Considering first the general category, function `blanks` has the syntax:




```
s = blanks(n)
```

It generates a string consisting of `n` blanks. Function `cellstr` creates a cell array of strings from a character array. One of the principal advantages of storing strings in cell arrays is that this approach eliminates the need to pad strings with blanks to create character arrays with rows of equal length (e.g., to perform string comparisons). The syntax




```
c = cellstr(S)
```

places the rows of the character array `S` into separate cells of `c`. Function `char` is used to convert back to a string matrix. For example, consider the string matrix

```
>> S = [' abc'; 'defg'; 'hi  '] % Note the blanks.
S =
     abc
    defg
    hi
```

Typing `whos S` at the prompt displays the following information:

```
>> whos S
  Name      Size      Bytes     Class     Attributes
  S         3x4       24        char
```

Note in the first command line that the third string in `S` has trailing blanks (all rows in a string matrix must have the same number of characters). Note also that no quotes enclose the strings in the output because `S` is a character

[†]Some of the string functions discussed in this section were introduced in earlier chapters.

TABLE 13.3 MATLAB string-manipulation functions.

Category	Function Name	Explanation
General	`blanks`	String of blanks.
	`cellstr`	Create a cell array of strings from a character array. Use function `char` to convert back to a character string.
	`char`	Create character array (string).
	`deblank`	Remove trailing blanks.
	`eval`	Execute string with MATLAB expression.
String tests	`iscellstr`	True for cell array of strings.
	`ischar`	True for character array.
	`isletter`	True for letters of the alphabet.
	`isspace`	True for whitespace characters.
String operations	`lower`	Convert string to lowercase.
	`regexp`	Match regular expression.
	`regexpi`	Match regular expression, ignoring case.
	`regexprep`	Replace string using regular expression.
	`strcat`	Concatenate strings.
	`strcmp`	Compare strings (see Section 2.10.6).
	`strcmpi`	Compare strings, ignoring case.
	`strfind`	Find one string within another.
	`strjust`	Justify string.
	`strmatch`	Find matches for string. (Use of `strcmp`, `strcmpi`, `strncmp`, or `strcnpi` is preferred because they are faster.)
	`strncmp`	Compare first n characters of strings.
	`strncmpi`	Compare first n characters, ignoring case.
	`strread`	Read formatted data from a string. See Section 2.10.6 for a detailed explanation.
	`strrep`	Replace a string within another.
	`strtok`	Find token in string.
	`strvcat`	Concatenate strings vertically.
	`upper`	Convert string to uppercase.
String to number conversion	`double`	Convert string to numeric codes.
	`int2str`	Convert integer to string.
	`mat2str`	Convert matrix to a string suitable for processing with the `eval` function.
	`num2str`	Convert number to string.
	`sprintf`	Write formatted data to string.
	`str2double`	Convert string to double-precision value.
	`str2num`	Convert string to number (see Section 2.10.6)
	`sscanf`	Read string under format control.
Base number conversion	`base2dec`	Convert base B string to decimal integer. For example, `base2dec('213', 3)` converts 212_3 to decimal, returning 23.
	`bin2dec`	Convert decimal integer to binary string.
	`dec2base`	Convert decimal integer to base B string.
	`dec2bin`	Convert decimal integer to binary string.
	`dec2hex`	Convert decimal integer to hexadecimal string.
	`hex2dec`	Convert hexadecimal string to decimal integer.
	`hex2num`	Convert IEEE hexadecimal to double-precision number.

array. The following command returns a 3×1 cell array (note that the third string has no trailing blanks):

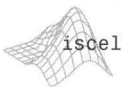
```
>> C = cellstr(S)
C =
    ' abc'
    'defg'
    'hi'

>> whos C
    Name      Size      Bytes     Class Attributes
    C         3x1       200       cell
```

where, for example, `C(1) = ' abc'` and `C{1} = abc`. Note that quotes appear around the strings when using `C(1)`.

```
>> Z = char(C)
Z =
     abc
    defg
    hi
```



Function `eval` evaluates a string that contains a MATLAB expression. The call `eval(expression)` executes `expression`, a string containing any valid MATLAB expression. For example, if `t` is the character string `t = '3^2'`, typing `eval(t)` returns a 9.



The next category of functions deals with string tests. A 1 is returned if the result of evaluating the function is true; otherwise the value returned is 0. Thus, in the preceding example, `iscellstr(C)` would return a 1 and `iscellstr(S)` would return a 0. Similar comments apply to the other functions in this category.

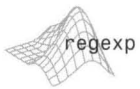
String operations are next. Functions `lower` (and `upper`) are self explanatory. They are discussed in Section 4.7.1. The next three functions deal with *regular expressions*,[†] which are sets of symbols and syntactic elements used commonly to match patterns of text. An example of the power of regular expressions is the use of the familiar wildcard symbol " * " in a file search. For instance, a search for image*.m in a typical search command window would return all the M-files that begin with the word "image." Another example of the use of regular expressions is in a search-and-replace function that searches for an instance of a given text string and replaces it with another. Regular

[†]Regular expressions can be traced to the work of American mathematician Stephen Kleene, who developed regular expressions as a notation for describing what he called "the algebra of regular sets."

expressions are formed using *metacharacters*, some of which are listed in Table 13.4. Several examples are given in the following paragraph.

Function `regexp` matches a regular expression. The syntax

```
idx = regexp(str, expr)
```



returns a row vector, `idx`, containing the indices (locations) of the substrings in `str` that match the regular expression string, `expr`. For example, suppose that `expr = 'b.*a'`. Then the expression `idx = regexp(str, expr)` would find matches in string `str` for any `b` that is followed by any character (as specified by the metacharacter "`.`") any number of times, including zero times (as specified by `*`), followed by an `a`. The indices of any locations in `str` meeting these conditions are stored in vector `idx`. If no such locations are found, then `idx` is returned as the empty matrix.

A few more examples of regular expressions for `expr` should clarify these concepts. The regular expression `'b.+ a'` would be as in the preceding example, except that "any number of times, including zero times" would be replaced by "one or more times." The expression `'b[0-9]'` means any `b` followed by any number from `0` to `9`; the expression `'b[0-9]*'` means any `b` followed by any number from `0` to `9` *any* number of times; and `'b[0-9]+'` means `b` followed by any number from `0` to `9` *one* or more times. For example, if `str = 'b0123c234bcd'`, the preceding three instances of `expr` would give the following results: `idx = 1`; `idx = [1 10]`; and `idx = 1`.

Metacharacters	Usage
`.`	Matches any one character.
`[ab...]`	Matches any one of the characters, (`a, b, ...`), contained within the brackets.
`[^ab...]`	Matches any character except those contained within the brackets.
`?`	Matches any character zero or one time.
`*`	Matches the preceding element zero or more times.
`+`	Matches the preceding element one or more times.
`{num}`	Matches the preceding element `num` times.
`{min, max}`	Matches the preceding element at least `min` times, but not more than `max` times.
`|`	Matches either the expression preceding or following the metacharacter `|`.
`^chars`	Matches when a string begins with `chars`.
`chars$`	Matches when a string ends with `chars`.
`\<chars`	Matches when a word begins with `chars`.
`chars>\`	Matches when a word ends with `chars`.
`\<word\>`	Exact `word` match.

TABLE 13.4
Some of the metacharacters used in regular expressions for matching. See the regular expressions help page for a complete list.

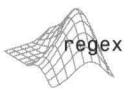
As an example of the use of regular expressions for recognizing object characteristics, suppose that the boundary of an object has been coded with a four-directional Freeman chain code [see Fig. 12.2(a)], stored in string `str`, so that `str = '0003003332222211111'`. Suppose also that we are interested in finding the locations in the string where the direction of travel turns from east (0) to south (3), and stays there for at least two increments, but no more than six increments. This is a "downward step" feature in the object, larger than a single transition (which may be due to noise). We can express these requirements in terms of the following regular expression:

```
>> expr = '0[3]{2,6}';
```

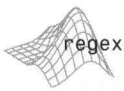
Then

```
>> idx = regexp(str, expr)
idx =
    6
```

The value of `idx` identified in this case the location where a `0` is followed by three `3`'s. More complex expressions are formed in a similar manner.



Function `regexpi` behaves in the manner just described for `regexp`, except that it ignores character (upper and lower) case. Function `regexprep`, with syntax




```
s = regexprep(str, expr, replace)
```

replaces with string `replace` all occurrences of the regular expression `expr` in string `str`. The new string is returned. If no matches are found, `regexprep` returns `str`, unchanged.

Function `strcat` has the syntax




```
C = strcat(S1, S2, S3, ...)
```

This function concatenates (horizontally) corresponding rows of the character arrays `S1`, `S2`, `S3`, and so on. All input arrays must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is a character array also. If any of the inputs is a cell array of strings, `strcat` returns a cell array of strings formed by concatenating corresponding elements of `S1`, `S2`, `S3`, and so on. The inputs must all have the same size (or any can be a scalar). Any of the inputs can be a character array also. Trailing spaces in character array inputs are ignored and do not appear in the output. This is not true for concatenated cell arrays of strings. To preserve trailing spaces the familiar concatenation syntax based on square brackets, `[S1 S2 S3 ...]`, should be used. For example,

```
>> a = 'hello ';   % Note the trailing blank space.
```

```
>> b = 'goodbye';
>> strcat(a, b)
ans =
    hellogoodbye

>> [a b]
ans =
    hello goodbye
```

Function `strvcat`, with syntax

```
S = strvcat(t1, t2, t3, ...)
```



forms the character array `S` containing the text strings (or string matrices) `t1, t2, t3, ...` as rows. Blanks are appended to each string as necessary to form a valid matrix. Empty arguments are ignored. For example, using the strings `a` and `b` from the preceding example,

```
>> strvcat(a, b)
ans =
    hello
    goodbye
```

Function `strcmp`, with syntax

```
k = strcmp(str1, str2)
```



compares the two strings in the argument and returns 1 (true) if the strings are identical. Otherwise it returns a 0 (false). A more general syntax is

```
K = strcmp(S, T)
```

where either `S` or `T` is a cell array of strings, and `K` is an array (of the same size as `S` and `T`) containing 1s for the elements of `S` and `T` that match, and 0s for the ones that do not. `S` and `T` must be of the same size (or one can be a scalar cell). Either one can be a character array also, with the proper number of rows. Function `strcmpi` performs the same operation as `strcmp`, but it ignores character case.



Function `strncmp`, with syntax

```
k = strncmp('str1', 'str2', n)
```



returns a logical true (1) if the first `n` characters of the strings `str1` and `str2` are the same, and returns a logical false (0) otherwise. Arguments `str1` and `str2` can be cell arrays of strings also. The syntax

```
R = strncmp(S, T, n)
```

where S and T can be cell arrays of strings, returns an array R the same size as S and T containing 1 for those elements of S and T that match (up to n characters), and 0 otherwise. S and T must be of the same size (or one can be a scalar cell). Either one can be a character array with the correct number of rows. The command strncmp is case sensitive. Any leading and trailing blanks in either of the strings are included in the comparison. Function strncmpi performs the same operation as strncmp, but ignores character case.



Function strfind, with syntax




```
I = strfind(str, pattern)
```

searches string str for occurrences of a *shorter* string, pattern, returning the starting index of each such occurrence in the double array, I. If pattern is not found in str, or if pattern is longer than str, then strfind returns the empty array, [ ].

Function strjust has the syntax




```
Q = strjust(A, direction)
```

where A is a character array, and direction can have the justification values 'right', 'left', and 'center'. The default justification is 'right'. The output array contains the same strings as A, but justified in the direction specified. Note that justification of a string implies the existence of leading and/or trailing blank characters to provide space for the specified operation. For instance, letting the symbol "□" represents a blank character, the string '□□abc' with two leading blank characters does not change under 'right' justification; becomes 'abc□□' with 'left' justification; and becomes the string '□abc□' with 'center' justification. Clearly, these operations have no effect on a string that does not contain any leading or trailing blanks.

Function strrep, with syntax



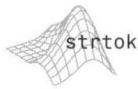

```
r = strrep('str1', 'str2', 'str3')
```

replaces all occurrences of the string str2 within string str1 with the string str3. If any of str1, str2, or str3 is a cell array of strings, this function returns a cell array the same size as str1, str2, and str3, obtained by performing a strrep using corresponding elements of the inputs. The inputs must all be of the same size (or any can be a scalar cell). Any one of the strings can be a character array also, with the correct number of rows. For example,

```
>> s = 'Image processing and restoration.';
>> str = strrep(s, 'processing', 'enhancement')
str =
    Image enhancement and restoration.
```

Function `strtok`, with syntax

```
t = strtok('str', delim)
```



returns the first token in the text string `str`, that is, the first set of characters before a delimiter in `delim` is encountered. Parameter `delim` is a vector containing delimiters (e.g., blanks, other characters, strings). For example,

```
>> str = 'An image is an ordered set of pixels';
>> delim = ' '; % Blank space.
>> t = strtok(str, delim)
t =
    An
```

Note that function `strtok` terminates after the first delimiter is encountered. (i.e., a blank character in the example just given). If we change `delim` to `delim = ['x']`, then the output becomes

```
>> t = strtok(str, delim)
t =
    An image is an ordered set of pi
```

The next set of functions in Table 13.2 deals with conversions between strings and numbers. Function `int2str`, with syntax

```
str = int2str(N)
```

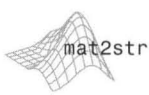


converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion. For example, `int2str(2 + 3.2)` is the *string* `'5'`. For matrix or vector inputs, `int2str` returns a string matrix:

```
>> str = int2str(eye(3))
ans =
    1   0   0
    0   1   0
    0   0   1

>> class(str)
ans =
    char
```

Function `mat2str`, with syntax

```
str = mat2str(A)
```



converts matrix A into a string, suitable for input to the `eval` function, using full precision. Using the syntax

```
str = mat2str(A, n)
```

converts matrix A using n digits of precision. For example, consider the matrix

```
>> A = [1 2; 3 4] % Note the space after the semicolon.
A =
    1   2
    3   4
```

The statement

```
>> b = mat2str(A)
```

produces

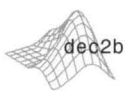
```
b =
    [1 2; 3 4]
```

where b is a string of 9 characters, including the square brackets, spaces, and a semicolon (the semicolon is a row terminator and any spaces after it are deleted by function `mat2str`). The command

```
>> eval(mat2str(A))
```

reproduces A. The other functions in this category have similar interpretations.

The last category in Table 13.2 deals with base number conversions. For example, function `dec2base`, with syntax




```
str = dec2base(d, base)
```

converts the decimal integer d to the specified base, where d must be a non-negative integer smaller than 2^52, and base must be an integer between 2 and 36. The returned argument str is a string. For example, the following command converts 23_{10} to base 2 and returns the result as a string:

```
>> str = dec2base(23, 2)
str =
    10111

>> class(str)
ans =
    char
```

Using the syntax

```
str = dec2base(d, base, n)
```

produces a representation with at least n digits.

13.4.2 String Matching

In addition to the string matching and comparing functions in Table 13.3, it is useful to have measures of similarity that behave similarly to the distance measures discussed in Section 13.2. We illustrate this approach using a measure defined as follows.

Suppose that two region boundaries, a and b, are coded into strings $a_1a_2 \ldots a_m$ and $b_1b_2 \ldots b_n$, respectively. Let α denote the number of matches between these two strings, where a match is said to occur in the kth position if $a_k = b_k$. The number of symbols that do not match is

$$\beta = \max(|a|, |b|) - a$$

where $|\arg|$ is the length (number of symbols) of the string in the argument. It can be shown that $\beta = 0$ if and only if a and b are identical strings.

A useful measure of similarity between a and b is the ratio

$$R = \frac{\alpha}{\beta} = \frac{\alpha}{\max(|a|, |b|) - a}$$

This measure, proposed by Sze and Yang [1981], is infinite for a perfect match and 0 when none of the corresponding symbols in a and b match (α is 0 in this case).

Because matching is performed between corresponding symbols, it is required that all strings be "registered" in some position-independent manner in order for this method to make sense. One way to register two strings is to shift one string with respect to the other until a maximum value of R is obtained. This, and other similar matching strategies, can be developed using the string operations in Table 13.3. Typically, a more efficient approach is to define the same starting point for all strings based on normalizing the boundaries with respect to size and orientation before their string representation is extracted. This approach is illustrated in Example 13.3.

The following M-function computes the preceding measure of similarity for two character strings.

strsimilarity

```
function R = strsimilarity(a, b)
%STRSIMILARITY Computes a similarity measure between two strings.
%   R = STRSIMILARITY(A, B) computes the similarity measure, R,
%   defined in Section 13.4.2 for strings A and B. The strings do
%   not have to be of the same length, but only one of the strings
%   can have blanks, and these must be trailing blanks. Blanks are
%   not counted when computing the length of the strings for use in
%   the similarity measure.

% Verify that a and b are character strings.
```

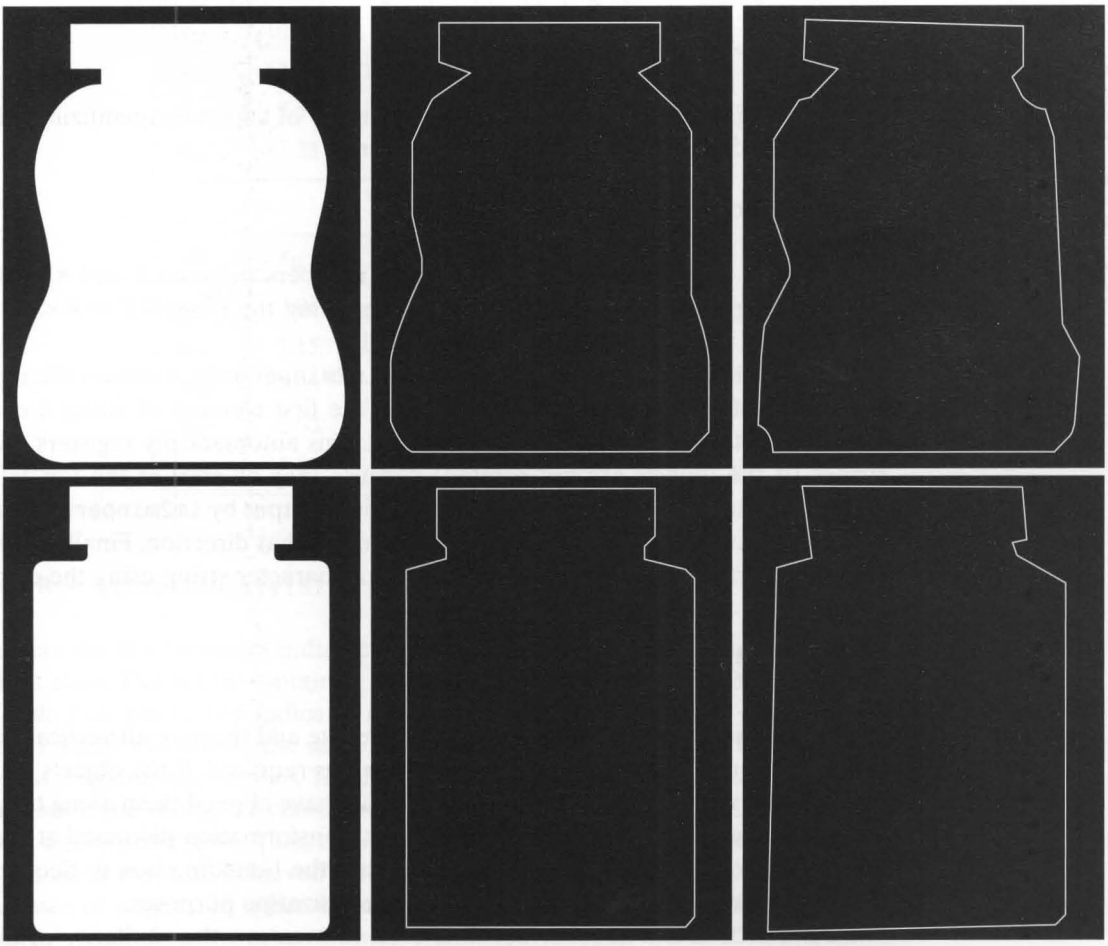
```
if ~ischar(a) || ~ischar(b)
   error('Inputs must be character strings.')
end

% Work with horizontal strings.
a = a(:)';
b = b(:)';

% Find any blank spaces.
I = find(a == ' ');
J = find(b == ' ');
LI = numel(I); % LI and LJ are used later.
LJ = numel(J);
% Check to see if one of the strings is blank, in which case R = 0.
if LI == length(a) || LJ == length(b)
   R = 0;
   return
end

if (LI ~= 0 && I(1) == 1) ||(LJ ~= 0 && J(1) == 1)
   error('Strings cannot contain leading blanks.')
end

if LI ~= 0 && LJ ~= 0
   error('Only one of the strings can contain blanks.')
end

% Pad the end of the shorter string.
La = length(a);
Lb = length(b);
if LI == 0 && LJ == 0
   if La > Lb
      b = [b, blanks(La - Lb)];
   else
      a = [a, blanks(Lb - La)];
   end
elseif isempty(J)
   Lb = length(b) - length(J);
   b = [b, blanks(La - Lb - LJ)];
else
   La = length(a) - length(I);
   a = [a, blanks(Lb - La - LI)];
end

% Compute the similarity measure.
I = find(a == b);
alpha = numel(I);
den = max(La, Lb) - alpha;
if den == 0
   R = Inf;
else
```

```
    R = alpha/den;
end
```

EXAMPLE 13.3: Object recognition based on string matching.

■ Figures 13.3(a) and (d) show silhouettes of two samples of container bottles whose principal shape difference is the curvature of their sides. For purposes of differentiation, objects with the curvature characteristics of Fig. 13.3(a) are said to be from class 1. Objects with straight sides are said to be from class 2. The images are of size 372×288 pixels.

To illustrate the effectiveness of measure R for differentiating between objects of classes 1 and 2, the boundaries of the objects were approximated by minimum-perimeter polygons using function `im2minperpoly` (see Section



a b c
d e f

FIGURE 13.3 (a) An object. (b) Its minimum perimeter polygon obtained using function `im2minperpoly` with a cell size of 8. (c) A noisy boundary. (d)-(f) The same sequence for another object.

12.2.2) with a cell size of 8. Figures 13.3(b) and (e) show the results. Then noise was added to the coordinates of each vertex of the polygons using function `randvertex` (see Appendix C), which has the syntax

`randvertex`

```
[xn, yn] = randvertex(x, y, npix)
```

where `x` and `y` are column vectors containing the coordinates of the vertices of a polygon, `xn` and `yn` are the corresponding noisy coordinates, and `npix` is the maximum number of pixels by which a coordinate is allowed to be displaced in either direction. Five sets of noisy vertices were generated for each class using `npix = 5`. Figures 13.3(c) and (f) show typical results.

Strings of symbols were generated for each class by coding the interior angles of the polygons using function `polyangles` (see Appendix C):

`polyangles`

```
>> angles = polyangles(x, y);
```

The `x` and `y` inputs to function `polyangles` are vectors containing the x- and y-coordinates of the vertices of a polygon, ordered in the clockwise direction. The output is a vector containing the corresponding interior angles, in degrees.

Then a string, `s`, was generated from a given array of angles by quantizing the angles into 45° increments, using the statement

```
>> s = floor(angles/45) + 1;
```

This yielded a string whose elements were numbers between 1 and 8, with 1 designating the range $0° \leq \theta < 45°$, 2 designating the range $45° \leq \theta < 90°$, and so forth, where θ denotes an interior angle.

Because the first vertex in the output of `im2minperpoly` is always the top, left vertex of the boundary of the input, `B`, the first element of string `s` corresponds to the interior angle of that vertex. This automatically registers the strings (if the objects are not rotated) because they all start at the top, left vertex in all images. The direction of the vertices output by `im2minperpoly` is counterclockwise, so the elements of `s` also are in that direction. Finally, each `s` was converted from a string of integers to a character string using the command

```
>> s = int2str(s);
```

In this example the objects are of comparable size and they are all vertical, so normalization of neither size nor orientation was required. If the objects had been of arbitrary size and orientation, we could have aligned them along their principal directions by using the eigenvector transformation discussed at the end of Section 12.5. Then we could have used the bounding box in Section 12.4.1 to obtain the object dimensions for normalization purposes.

First, function `strsimilarity` was used to measure the similarity of all strings of class 1 between themselves. For instance, to compute the similarity between the first and second strings of class 1 we used the command

R	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
s_{11}	Inf				
s_{12}	9.33	Inf			
s_{13}	26.25	12.31	Inf		
s_{14}	16.36	9.33	14.16	Inf	
s_{15}	22.22	14.17	14.01	19.02	Inf

TABLE 13.5
Values of similarity measure R between the strings of class 1. (All values are $\times 10$.)

R	s_{21}	s_{22}	s_{23}	s_{24}	s_{25}
s_{21}	Inf				
s_{22}	10.00	Inf			
s_{23}	13.33	13.33	Inf		
s_{24}	7.50	13.31	18.00	Inf	
s_{25}	13.33	7.51	18.12	10.01	Inf

TABLE 13.6
Values of similarity measure R between the strings of class 2. (All values are $\times 10$.)

R	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
s_{21}	2.03	0.01	1.15	1.17	0.75
s_{22}	1.15	1.61	1.16	0.75	2.07
s_{23}	2.08	1.15	2.08	2.06	2.08
s_{24}	1.60	1.62	1.59	1.14	2.61
s_{25}	1.61	0.36	0.74	1.60	1.16

TABLE 13.7
Values of similarity measure R between the strings of classes 1 and 2. (All values are $\times 10$.)

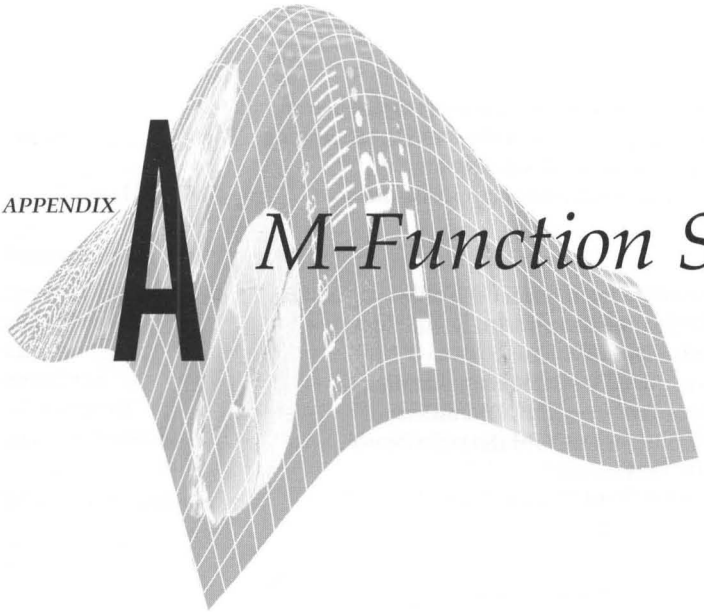
```
>> R = strsimilarity(s11, s12);
```

where the first subscript indicates class and the second a string number within that class. The results obtained using five typical strings are summarized in Table 13.5, where `Inf` indicates infinity (i.e., a perfect match, as discussed earlier). Table 13.6 shows the same type of computation involving five strings of class 2 against themselves. Table 13.7 shows values of the similarity measure between the strings of class 1 and class 2. Note that the values in this table are significantly lower than the entries in the two preceding tables, indicating that the R measure achieved a high degree of discrimination between the two classes of objects. In other words, measuring the similarity of strings against members of their own class showed significantly larger values of R, indicating a closer match than when strings were compared to members of the opposite class. ■

Summary

Starting with Chapter 10, our treatment of digital image processing began a transition from processes whose outputs are images to processes whose outputs are attributes extracted from those images. Although the material in the present chapter is introductory in nature, the topics covered are fundamental to understanding the state of the art in object recognition. As mentioned in Section 1.2 at the onset of our journey, recognition of individual objects is a logical place at which to conclude this book.

Having finished studying the material in the preceding thirteen chapters, you are now in the position of being able to master the fundamentals of how to prototype software solutions of image-processing problems using MATLAB and Image Processing Toolbox functions. What is even more important, the background and numerous new functions developed in the book constitute a basic blueprint for how to extend the power of MATLAB and the toolbox. Given the task-specific nature of most imaging problems, a clear understanding of this material enhances significantly your chances of arriving at successful solutions in a broad spectrum of image processing application areas.



APPENDIX A

M-Function Summary

Preview

Section A.1 of this appendix contains a listing by name of all the functions in the Image Processing Toolbox, and all the new (custom) functions developed in the preceding chapters. The latter functions are referred to as *DIPUM* functions, a term derived from the first letter of the words in the title of the book. Section A.2 lists the MATLAB functions used throughout the book. All page numbers listed refer to pages in the book, indicating where a function is first used and illustrated. In some instances, more than one location is given, indicating that the function is explained in different ways, depending on the application. Use of a gray dash "—" in the page reference indicates a toolbox function not used in the book; information about them can be obtained in the product documentation. All MATLAB functions listed in Section A.2 are used in the book. Each page number in that section identifies the first use of the MATLAB function indicated.

The following functions are grouped loosely in categories similar to those found in Image Processing Toolbox documentation. A new category (e.g., wavelets) was created in cases for which no toolbox category exists (e.g., wavelets).

A.1 Image Processing Toolbox and DIPUM Functions

The following functions are grouped loosely into categories similar to those found in Image Processing Toolbox documentation.

Function category and Name	Description	Pages
Image display and exploration		
`ice` (DIPUM)	Interactive Color Editor.	352, 727
`immovie`	Make movie from multiframe image.	—
`implay`	Play movies, videos, or image sequences.	407, 474
`imshow`	Display image in Handle Graphics figure.	18

`imtool`	Display image in the Image Tool.	19
`montage`	Display multiple image frames as rectangular montage.	474
`rgbcube` (DIPUM)	Displays an RGB cube on the MATLAB desktop.	319
`subimage`	Display multiple images in single figure.	—
`warp`	Display image as texture-mapped surface	—

Image file I/O

`analyze75info`	Read metadata from header file of Mayo Analyze 7.5 data set.	—
`analyze75read`	Read image file of Mayo Analyze 7.5 data set.	—
`dicomanon`	Anonymize DICOM file.	—
`dicomdict`	Get or set active DICOM data dictionary.	—
`dicominfo`	Read metadata from DICOM message.	—
`dicomlookup`	Find attribute in DICOM data dictionary.	—
`dicomread`	Read DICOM image.	—
`dicomuid`	Generate DICOM Unique Identifier.	—
`dicomwrite`	Write images as DICOM files.	—
`hdrread`	Read Radiance HDR image.	—
`hdrwrite`	Write Radiance HDR image.	—
`makehdr`	Create high dynamic range image.	—
`interfileinfo`	Read metadata from Interfile files.	—
`interfileread`	Read images from Interfile files.	—
`isnitf`	Check if file is NITF.	—
`movie2tifs` (DIPUM)	Creates a multiframe TIFF file from a MATLAB movie.	475
`nitfinfo`	Read metadata from NITF file.	—
`nitfread`	Read NITF image.	—
`seq2tifs` (DIPUM)	Creates a multi-frame TIFF file from a MATLAB sequence.	475
`tifs2movie` (DIPUM)	Create a MATLAB movie from a multiframe TIFF file.	475
`tifs2seq` (DIPUM)	Create a MATLAB sequence from a multi-frame TIFF file.	475

Image arithmetic

`imabsdiff`	Absolute difference of two images.	—
`imcomplement`	Complement image.	83, 331
`imlincomb`	Linear combination of images.	50
`ippl`	Check for presence of Intel Performance Primitives Library (IPPL).	—

Geometric transformations

`checkerboard`	Create checkerboard image.	238
`findbounds`	Find output bounds for spatial transformation.	—
`fliptform`	Flip input and output roles of TFORM structure.	—
`imcrop`	Crop image.	—
`impyramid`	Image pyramid reduction and expansion.	—
`imresize`	Resize image.	—
`imrotate`	Rotate image.	291, 659
`imtransform`	Apply 2-D spatial transformation to image.	289
`imtransform2` (DIPUM)	2-D image transformation with fixed output location.	298
`makeresampler`	Create resampling structure.	—
`maketform`	Create spatial transformation structure (TFORM).	279, 309
`pixeldup` (DIPUM)	Duplicates pixels of an image in both directions.	238
`pointgrid` (DIPUM)	Points arranged on a grid.	282

`reprotate` (DIPUM)	Rotate image repeatedly.	303
`tformarray`	Apply spatial transformation to N-D array.	—
`tformfwd`	Apply forward spatial transformation.	281
`tforminv`	Apply inverse spatial transformation.	281
`vistform` (DIPUM)	Visualization transformation effect on set of points.	283

Image registration

`cpstruct2pairs`	Convert CPSTRUCT to control point pairs.	—
`cp2tform`	Infer spatial transformation from control point pairs.	307
`cpcorr`	Tune control point locations using cross-correlation.	—
`cpselect`	Control Point Selection Tool.	306
`normxcorr2`	Normalized two-dimensional cross-correlation.	313, 683
`visreg` (DIPUM)	Visualize registered images.	308

Pixel values and statistics

`corr2`	2-D correlation coefficient.	—
`imcontour`	Create contour plot of image data.	—
`imhist`	Display histogram of image data.	94
`impixel`	Pixel color values.	—
`improfile`	Pixel-value cross-sections along line segments.	—
`localmean` (DIPUM)	Computes an array of local means.	572
`mean2`	Average or mean of matrix elements.	76, 92
`regionprops`	Measure properties of image regions (blob analysis).	642
`statmoments` (DIPUM)	Computes statistical central moments of image histogram.	225
`std2`	Standard deviation of matrix elements.	—

Image analysis

`bayesgauss` (DIPUM)	Bayes classifier for Gaussian patterns.	685
`bound2eight` (DIPUM)	Convert 4-connected boundary to 8-connected boundary.	605
`bound2four` (DIPUM)	Convert 8-connected boundary to 4-connected boundary.	605
`bound2im` (DIPUM)	Converts a boundary to an image.	600
`bsubsamp` (DIPUM)	Subsample a boundary.	605
`bwboundaries` (DIPUM)	Trace region boundaries in binary image.	599
`bwtraceboundary`	Trace object in binary image.	—
`colorgrad` (DIPUM)	Computes the vector gradient of an RGB image.	369
`colorseg` (DIPUM)	Performs segmentation of a color image.	373
`connectpoly` (DIPUM)	Connects vertices of a polygon.	605
`cornermetric`	Create corner metric matrix from image.	638
`cornerprocess` (DIPUM)	Processes the output of function `cornermetric`.	638
`diameter` (DIPUM)	Measure diameter and related properties of image regions.	626
`edge`	Find edges in intensity image.	542
`fchcode` (DIPUM)	Computes the Freeman chain code of a boundary.	607
`frdescp` (DIPUM)	Computes Fourier descriptors.	629
`ifrdescp` (DIPUM)	Computes inverse Fourier descriptors.	629
`im2minperpoly` (DIPUM)	Minimum perimeter polygon.	617
`imstack2vectors` (DIPUM)	Extracts vectors from an image stack.	663
`invmoments` (DIPUM)	Compute invariant moments of image.	658
`hough`	Hough transform.	553
`houghlines`	Extract line segments based on Hough transform.	555

houghpeaks	Identify peaks in Hough transform.	555
localthresh (DIPUM)	Local thresholding.	573
mahalanobis (DIPUM)	Computes the Mahalanobis distance.	678
movingthresh (DIPUM)	Image segmentation using a moving average threshold.	576
otsuthresh (DIPUM)	Otsu's optimum threshold given a histogram.	564
polyangles (DIPUM)	Computes internal polygon angles.	704
principalcomps (DIPUM)	Principal-component vectors and related quantities.	664
qtdecomp	Quadtree decomposition.	584
qtgetblk	Get block values in quadtree decomposition.	584
qtsetblk	Set block values in quadtree decomposition.	—
randvertex (DIPUM)	Adds random noise to the vertices of a polygon.	704
regiongrow (DIPUM)	Perform segmentation by region growing.	580
signature (DIPUM)	Computes the signature of a boundary.	620
specxture (DIPUM)	Computes spectral texture of an image.	655
splitmerge (DIPUM)	Segment an image using a split-and-merge algorithm.	585
statxture (DIPUM)	Computes statistical measures of texture in an image.	645
strsimilarity (DIPUM)	Computes a similarity measure between two strings.	701
x2majoraxis (DIPUM)	Aligns coordinate x with the major axis of a region.	628

Image compression

compare (DIPUM)	Computes and displays the error between two matrices.	423
cv2tifs (DIPUM)	Decodes a TIFS2CV compressed image sequence.	483
huff2mat (DIPUM)	Decodes a Huffman encoded matrix.	440
huffman (DIPUM)	Builds a variable-length Huffman code for a symbol source.	429
im2jpeg (DIPUM)	Compresses an image using a JPEG approximation.	457
im2jpeg2k (DIPUM)	Compresses an image using a JPEG 2000 approximation.	466
imratio (DIPUM)	Computes the ratio of the bytes in two images/variables.	421
jpeg2im (DIPUM)	Decodes an IM2JPEG compressed image.	461
jpeg2k2im (DIPUM)	Decodes an IM2JPEG2K compressed image.	469
lpc2mat (DIPUM)	Decompresses a 1-D lossless predictive encoded matrix.	451
mat2huff (DIPUM)	Huffman encodes a matrix.	436
mat2lpc (DIPUM)	Compresses a matrix using 1-D lossles predictive coding.	450
ntrop (DIPUM)	Computes a first-order estimate of the entropy of a matrix.	426
quantize (DIPUM)	Quantizes the elements of a UINT8 matrix.	454
showmo (DIPUM)	Displays the motion vectors of a compressed image sequence.	483
tifs2cv (DIPUM)	Compresses a multi-frame TIFF image sequence.	480
unravel (DIPUM)	Decodes a variable-length bit stream.	442

Image deblurring

deconvblind	Deblur image using blind deconvolution.	250
deconvlucy	Deblur image using Lucy-Richardson method.	248
deconvreg	Deblur image using regularized filter.	245
deconvwnr	Deblur image using Wiener filter.	241
edgetaper	Taper edges using point-spread function.	242
otf2psf	Convert optical transfer function to point-spread function.	—
psf2otf	Convert point-spread function to optical transfer function.	—

Image enhancement

adapthisteq	Contrast-limited Adaptive Histogram Equalization (CLAHE).	107
adpmedian (DIPUM)	Perform adaptive median filtering.	235

decorrstretch	Apply decorrelation stretch to multichannel image.	—
gscale (DIPUM)	Scales the intensity of the input image.	92
histeq	Enhance contrast using histogram equalization.	100
imadjust	Adjust image intensity values or color map.	82
medfilt2	2-D median filtering.	—
ordfilt2	2-D order-statistic filtering.	125
stretchlim	Find limits to contrast stretch an image.	84
intlut	Convert integer values using lookup table.	—
intrans (DIPUM)	Performs intensity (gray-level) transformations.	89
wiener2	2-D adaptive noise-removal filtering.	—

Image noise

imnoise	Add noise to image.	126, 211
imnoise2 (DIPUM)	Generates an array of random numbers with specified PDF.	216
imnoise3 (DIPUM)	Generates periodic noise.	221

Linear filtering

convmtx2	2-D convolution matrix.	—
dftfilt (DIPUM)	Performs frequency domain filtering.	179
fspecial	Create predefined 2-D filters.	120
imfilter	N-D filtering of multidimensional images.	114
spfilt (DIPUM)	Performs linear and nonlinear spatial filtering.	229

Linear 2-D filter design

bandfilter (DIPUM)	Computes frequency domain band filters.	199
cnotch (DIPUM)	Generates circularly symmetric notch filters.	203
freqz2	2-D frequency response.	181
fsamp2	2-D FIR filter using frequency sampling.	—
ftrans2	2-D FIR filter using frequency transformation.	—
fwind1	2-D FIR filter using 1-D window method.	—
fwind2	2-D FIR filter using 2-D window method.	—
hpfilter (DIPUM)	Computes frequency domain highpass filters.	195
lpfilter (DIPUM)	Computes frequency domain lowpass filters.	175,189
recnotch (DIPUM)	Generates rectangular notch (axes) filters.	205

Fuzzy logic

aggfcn (DIPUM)	Aggregation function for a fuzzy system.	149
approxfcn (DIPUM)	Approximation function.	152
bellmf (DIPUM)	Bell-shaped membership function.	145
defuzzify (DIPUM)	Output of fuzzy system.	149
fuzzyfilt (DIPUM)	Fuzzy edge detector.	162
fuzzysysfcn (DIPUM)	Fuzzy system function.	150
implfcns (DIPUM)	Implication functions for a fuzzy system.	147
lambdafcns (DIPUM)	Lambda functions for a set of fuzzy rules.	146
makefuzzyedgesys (DIPUM)	Script to make MAT-file used by FUZZYFILT.	161
onemf (DIPUM)	Constant membership function (one).	145
sigmamf (DIPUM)	Sigma membership function.	144
smf (DIPUM)	S-shaped membership function.	144
trapezmf (DIPUM)	Trapezoidal membership function.	143

`triangmf` (DIPUM)	Triangular membership function.	143
`truncgaussmf` (DIPUM)	Truncated Gaussian membership function.	145
`zeromf` (DIPUM)	Constant membership function (zero).	145

Image transforms

`dct2`	2-D discrete cosine transform.	—
`dctmtx`	Discrete cosine transform matrix.	
`fan2para`	Convert fan-beam projections to parallel-beam.	274
`fanbeam`	Fan-beam transform.	269
`idct2`	2-D inverse discrete cosine transform.	—
`ifanbeam`	Inverse fan-beam transform.	271
`iradon`	Inverse Radon transform.	263
`para2fan`	Convert parallel-beam projections to fan-beam.	275
`phantom`	Create head phantom image.	261
`radon`	Radon transform.	260

Neighborhood and block processing

`bestblk`	Optimal block size for block processing.	—
`blkproc`	Distinct block processing for image.	459
`col2im`	Rearrange matrix columns into blocks.	460
`colfilt`	Columnwise neighborhood operations.	118
`im2col`	Rearrange image blocks into columns.	460
`nlfilter`	General sliding-neighborhood operations.	—

Morphological operations (gray scale and binary images)

`conndef`	Default connectivity array.	—
`imbothat`	Bottom-hat filtering.	529
`imclearborder`	Suppress light structures connected to image border.	521
`imclose`	Morphologically close image.	501
`imdilate`	Dilate image.	492
`imerode`	Erode image.	500
`imextendedmax`	Extended-maxima transform.	—
`imextendedmin`	Extended-minima transform.	595
`imfill`	Fill image regions and holes.	521, 603
`imhmax`	H-maxima transform.	—
`imhmin`	H-minima transform.	531
`imimposemin`	Impose minima.	596
`imopen`	Morphologically open image.	501
`imreconstruct`	Morphological reconstruction.	518
`imregionalmax`	Regional maxima.	—
`imregionalmin`	Regional minima.	593
`imtophat`	Top-hat filtering.	529
`watershed`	Watershed transform.	590

Morphological operations (binary images)

`applylut`	Neighborhood operations using lookup tables.	507
`bwarea`	Area of objects in binary image.	—
`bwareaopen`	Morphologically open binary image (remove small objects).	—
`bwdist`	Distance transform of binary image.	589

bweuler	Euler number of binary image.	—
bwhitmiss	Binary hit-miss operation.	505
bwlabel	Label connected components in 2-D binary image.	515
bwlabeln	Label connected components in N-D binary image.	—
bwmorph	Morphological operations on binary image.	511
bwpack	Pack binary image.	
bwperim	Find perimeter of objects in binary image.	598
bwselect	Select objects in binary image.	—
bwulterode	Ultimate erosion.	—
bwunpack	Unpack binary image.	—
endpoints (DIPUM)	Computes end points of a binary image.	507
makelut	Create lookup table for use with APPLYLUT.	507

Structuring element (STREL) creation and manipulation

getheight	Get STREL height.	—
getneighbors	Get offset location and height of STREL neighbors.	—
getnhood	Get STREL neighborhood.	—
getsequence	Get sequence of decomposed STRELs.	497
isflat	True for flat STRELs.	—
reflect	Reflect STREL about its center.	492
strel	Create morphological structuring element (STREL).	494
translate	Translate STREL.	—

Texture analysis

entropy	Entropy of intensity image.	—
entropyfilt	Local entropy of intensity image.	—
graycomatrix	Create gray-level co-occurrence matrix.	648
graycoprops	Properties of gray-level co-occurrence matrix.	649
rangefilt	Local range of image.	—
specxture (DIPUM)	Computes spectral texture of an image.	655
statxture (DIPUM)	Computes statistical measures of texture in an image.	645
stdfilt	Local standard deviation of image.	572

Region-based processing

histroi (DIPUM)	Computes the histogram of an ROI in an image.	227
poly2mask	Convert region-of-interest polygon to mask.	—
roicolor	Select region of interest based on color.	—
roifill	Fill in specified polygon in grayscale image.	—
roifilt2	Filter region of interest.	—
roipoly	Select polygonal region of interest.	225

Wavelets

appcoef2	Extract 2-D approximation coefficients.	398
detcoef2	Extract 2-D detail coefficients.	398
dwtmode	Discrete wavelet transform extension mode.	387
waveback (DIPUM)	Computes inverse FWTs for multi-level decomposition.	409
wavecopy (DIPUM)	Fetches coefficients of a wavelet decomposition structure.	402
wavecut (DIPUM)	Zeroes coefficients in a wavelet decomposition structure.	401
wavedec2	Multilevel 2-D wavelet decomposition.	385

`wavedisplay` (DIPUM)	Display wavelet decomposition coefficients.	404
`wavefast` (DIPUM)	Computes the FWT of a '3-D extended' 2-D array.	391
`wavefilter` (DIPUM)	Create wavelet decomposition and reconstruction filters.	388
`wavefun`	Wavelet and scaling functions 1-D.	382
`waveinfo`	Information on wavelets.	382
`waverec2`	Multilevel 2-D wavelet reconstruction.	409
`wavework` (DIPUM)	is used to edit wavelet decomposition structures.	399
`wavezero` (DIPUM)	Zeroes wavelet transform detail coefficients.	415
`wfilters`	Wavelet filters.	381
`wthcoef2`	Wavelet coefficient thresholding 2-D.	398

Colormap manipulation

`cmpermute`	Rearrange colors in color map.	—
`cmunique`	Eliminate unneeded colors in color map of indexed image.	—
`imapprox`	Approximate indexed image by one with fewer colors.	321

Color space conversions

`applycform`	Apply device-independent color space transformation.	344
`hsi2rgb` (DIPUM)	Converts an HSI image to RGB.	338
`iccfind`	Search for ICC profiles by description.	—
`iccread`	Read ICC color profile.	347
`iccroot`	Find system ICC profile repository.	—
`iccwrite`	Write ICC color profile.	—
`isicc`	True for complete profile structure.	—
`lab2double`	Convert L*a*b* color values to double.	—
`lab2uint16`	Convert L*a*b* color values to uint16.	—
`lab2uint8`	Convert L*a*b* color values to uint8.	—
`makecform`	Create device-independent color space transformation structure (CFORM).	344
`ntsc2rgb`	Convert NTSC color values to RGB color space.	329
`rgb2hsi` (DIPUM)	Converts an RGB image to HSI.	337
`rgb2ntsc`	Convert RGB color values to NTSC color space.	328
`rgb2ycbcr`	Convert RGB color values to YCbCr color space.	329
`whitepoint`	XYZ color values of standard illuminants.	—
`xyz2double`	Convert XYZ color values to double.	—
`xyz2uint16`	Convert XYZ color values to uint16.	—
`ycbcr2rgb`	Convert YCbCr color values to RGB color space.	329

Array operations

`dftuv` (DIPUM)	Computes meshgrid frequency matrices.	186
`padarray`	Pad array.	118
`paddedsize` (DIPUM)	Computes padded sizes useful for FFT-based filtering.	174

Image types and type conversions

`demosaic`	Convert Bayer pattern encoded image to a true color image.	—
`dither`	Convert image using dithering.	324
`gray2ind`	Convert intensity image to indexed image.	325
`grayslice`	Create indexed image from intensity image by thresholding.	325
`graythresh`	Global image threshold using Otsu's method.	562

`im2bw`	Convert image to binary image by thresholding.	30
`im2double`	Convert image to double precision.	29
`im2int16`	Convert image to 16-bit signed integers.	—
`im2java2d`	Convert image to Java Buffered Image.	—
`im2single`	Convert image to single precision.	29
`im2uint8`	Convert image to 8-bit unsigned integers.	29
`im2uint16`	Convert image to 16-bit unsigned integers.	29
`ind2gray`	Convert indexed image to intensity image.	325
`label2rgb`	Convert label matrix to RGB image.	—
`mat2gray`	Convert matrix to intensity image.	30
`rgb2gray`	Convert RGB image or color map to grayscale.	326
`rgb2ind`	Convert RGB image to indexed image.	325
`tofloat` (DIPUM)	Convert image to floating point.	32
`tonemap`	Render high dynamic range image for viewing.	—

Toolbox preferences

`iptgetpref`	Get value of Image Processing Toolbox preference.	—
`iptsetpref`	Set value of Image Processing Toolbox preference.	291

Toolbox utility functions

`getrangefromclass`	Get dynamic range of image based on its class.	—
`intline`	Integer-coordinate line drawing.	606
`iptcheckconn`	Check validity of connectivity argument.	—
`iptcheckinput`	Check validity of array.	—
`iptcheckmap`	Check validity of color map.	—
`iptchecknargin`	Check number of input arguments.	—
`iptcheckstrs`	Check validity of text string.	—
`iptnum2ordinal`	Convert positive integer to ordinal string.	—

Modular interactive tools

`imageinfo`	Image Information tool.	—
`imcontrast`	Adjust Contrast tool.	—
`imdisplayrange`	Display Range tool.	—
`imdistline`	Draggable Distance tool.	—
`imgetfile`	Open Image dialog box.	—
`impixelinfo`	Pixel Information tool.	—
`impixelinfoval`	Pixel Information tool without text label.	—
`impixelregion`	Pixel Region tool.	—
`impixelregionpanel`	Pixel Region tool panel.	—
`imputfile`	Save Image dialog box.	—
`imsave`	Save Image tool.	—

Navigational tools for image scroll panel

`imscrollpanel`	Scroll panel for interactive image navigation.	—
`immagbox`	Magnification box for scroll panel.	—
`imoverview`	Overview tool for image displayed in scroll panel.	—
`imoverviewpanel`	Overview tool panel for image displayed in scroll panel.	—

Utility functions for interactive tools

`axes2pix`	Convert axes coordinate to pixel coordinate.	—
`getimage`	Get image data from axes.	—
`getimagemodel`	Get image model object from image object.	—
`imagemodel`	Image model object.	—
`imattributes`	Information about image attributes.	—
`imhandles`	Get all image handles.	—
`imgca`	Get handle to current axes containing image.	—
`imgcf`	Get handle to current figure containing image.	—
`imellipse`	Create draggable, resizable ellipse.	—
`imfreehand`	Create draggable freehand region.	—
`imline`	Create draggable, resizable line.	—
`impoint`	Create draggable point.	—
`impoly`	Create draggable, resizable polygon.	—
`imrect`	Create draggable, resizable rectangle.	—
`iptaddcallback`	Add function handle to callback list.	—
`iptcheckhandle`	Check validity of handle.	—
`iptgetapi`	Get Application Programmer Interface (API) for handle.	—
`iptGetPointerBehavior`	Retrieve pointer behavior from HG object.	—
`ipticondir`	Directories containing IPT and MATLAB icons.	—
`iptPointerManager`	Install mouse pointer manager in figure.	—
`iptremovecallback`	Delete function handle from callback list.	—
`iptSetPointerBehavior`	Store pointer behavior in HG object.	—
`iptwindowalign`	Align figure windows.	—
`makeConstrainToRectFcn`	Create rectangularly bounded position constraint function.	—
`truesize`	Adjust display size of image.	—

Interactive mouse utility functions

`getline`	Select polyline with mouse.	—
`getpts`	Select points with mouse.	—
`getrect`	Select rectangle with mouse.	—

Miscellaneous functions

`conwaylaws` (DIPUM)	Applies Conway's genetic laws to a single pixel.	509
`i2percentile` (DIPUM)	Computes a percentile given an intensity value.	567
`iseven` (DIPUM)	Determines which elements of an array are even numbers.	203
`isodd` (DIPUM)	Determines which elements of an array are odd numbers.	203
`manualhist` (DIPUM)	Generates a two-mode histogram interactively.	105
`timeit` (DIPUM)	Measure time required to run function.	66
`percentile2i` (DIPUM)	Computes an intensity value given a percentile.	567
`tofloat` (DIPUM)	Converts input to single-precision floating point.	*32*
`twomodegauss` (DIPUM)	Generates a two-mode Gaussian function.	104

A.2 MATLAB Functions

The following MATLAB functions, listed alphabetically, are used in the book.

MATLAB Function	Description	Pages
A		
`abs`	Absolute value.	168
`all`	True if all elements of a vector are nonzero.	53
`angle`	Phase angle.	171
`annotation`	Creates an annotation object.	102
`ans`	Most recent answer.	55
`any`	True if any element of a vector is nonzero.	53
`atan2`	Four quadrant inverse tangent.	170
`autumn`	Shades of red and yellow color map.	324
`axis`	Control axis scaling and appearance.	96
`axis`	Control axis scaling and appearance.	191
B		
`bar`	Bar graph.	95
`base2dec`	Convert base B string to decimal integer.	693
`bin2dec`	Convert binary string to decimal integer.	438
`bin2dec`	Convert binary string to decimal integer.	693
`blanks`	String of blanks.	692
`bone`	Gray-scale with a tinge of blue color map.	324
`break`	Terminate execution of WHILE or FOR loop.	61
`bsxfun`	Binary singleton expansion function.	676
C		
`cart2pol`	Transform Cartesian to polar coordinates.	621
`cat`	Concatenate arrays.	319
`catch`	Begin CATCH block.	58
`ceil`	Round towards plus infinity.	171
`cell`	Create cell array.	431
`celldisp`	Display cell array contents.	75, 431
`cellfun`	Apply a function to each cell of a cell array.	75
`cellplot`	Display graphical depiction of cell array.	431
`cellstr`	Create cell array of strings from character array.	692
`char`	Create character array (string).	26, 73, 693
`circshift`	Shift array circularly.	605
`colon`	Colon operator (:) for forming vectors and indexing.	33
`colorcube`	Enhanced color-cube color map.	324
`colormap`	Color look-up table.	191, 323
`computer`	Computer type.	55
`continue`	Pass control to the next iteration of FOR or WHILE loop.	62
`conv2`	Two dimensional convolution.	394

`cool`	Shades of cyan and magenta color map.	324
`copper`	Linear copper-tone color map.	323
`cumsum`	Cumulative sum of elements.	101

D

`deblank`	Remove trailing blanks.	693
`dec2base`	Convert decimal integer to base B string.	700
`dec2bin`	Convert decimal integer to a binary string.	436
`dec2hex`	Convert decimal integer to hexadecimal string.	693
`diag`	Diagonal matrices and diagonals of a matrix.	374
`diff`	Difference and approximate derivative.	529
`disp`	Display array.	71
`dither`	Convert image using dithering.	323
`double`	Convert to double precision.	26

E

`edit`	Edit M-file.	46
`eig`	Eigenvalues and eigenvectors.	665
`else`	Used with IF.	58
`elseif`	IF statement condition.	58
`end`	Terminate scope of FOR, WHILE, SWITCH, TRY, and IF statements.	34
`eps`	Spacing of floating point numbers.	55
`error`	Display message and abort function.	59
`eval`	Execute string with MATLAB expression.	694
`eye`	Identity matrix.	44

F

`false`	False array.	44, 587
`fft2`	Two-dimensional discrete Fourier Transform.	168
`fftshift`	Shift zero-frequency component to center of spectrum.	169
`figure`	Create figure window.	19
`filter`	One-dimensional digital filter.	575
`find`	Find indices of nonzero elements.	215
`fix`	Round towards zero.	152
`flag`	Alternating red, white, blue, and black color map.	324
`fliplr`	Flip matrix in left/right direction.	262
`flipud`	Flip matrix in up/down direction.	262
`floor`	Round towards minus infinity.	171
`for`	Repeat statements a specific number of times.	59
`format`	Set output format.	56
`fplot`	Plot function.	98
`full`	Convert sparse matrix to full matrix.	43

G

`gca`	Get handle to current axis.	96
`gcf`	Get handle to current figure.	737
`get`	Get object properties.	56, 353

`getfield`	Get structure field contents.	737
`global`	Define global variable.	430
`gray`	Linear gray-scale color map.	324
`grid`	Grid lines.	191
`gui_mainfcn`	Support function for creation and callback dispatch of GUIDE GUIs.	730
`guidata`	Store or retrieve application data.	736
`guide`	Open the GUI Design Environment.	725

H

`help`	Display help text in Command Window.	46
`hex2dec`	Convert hexadecimal string to decimal integer.	693
`hex2num`	Convert IEEE hexadecimal string to double precision number.	693
`hist`	Histogram.	220
`histc`	Histogram count.	437
`hold`	Hold current graph.	98
`hot`	Black-red-yellow-white color map.	324
`hsv`	Hue-saturation-value color map.	324
`hsv2rgb`	Convert hue-saturation-value colors to red-green-blue.	330
`hypot`	Robust computation of the square root of the sum of squares.	187
`hypot`	Robust computation of the square root of the sum of squares.	270

I

`i`	Imaginary unit.	55
`if`	Conditionally execute statements.	58
`ifft2`	Two-dimensional inverse discrete Fourier transform.	172
`ifftshift`	Inverse FFT shift.	170
`im2frame`	Convert indexed image into movie format.	473
`imag`	Complex imaginary part.	170
`imfinfo`	Information about graphics file.	23
`imread`	Read image from graphics file.	15
`imwrite`	Write image to graphics file.	21, 473
`ind2rgb`	Convert indexed image to RGB image.	326
`ind2sub`	Multiple subscripts from linear index.	40
`inpolygon`	True for points inside or on a polygonal region.	616
`input`	Prompt for user input.	72
`int16`	Convert to signed 16-bit integer.	26
`int2str`	Convert integer to string.	699
`int32`	Convert to signed 32-bit integer.	26
`int8`	Convert to signed 8-bit integer.	26
`interpn`	N-D interpolation (table lookup).	153
`interp1`	1-D interpolation (table lookup).	86
`interp1q`	Quick 1-D linear interpolation.	351
`iscell`	True for cell array.	54
`iscellstr`	True for cell array of strings.	54, 694
`ischar`	True for character array (string).	54, 693
`isempty`	True for empty array.	54
`isequal`	True if arrays are numerically equal.	54
`isfield`	True if field is in structure array.	54
`isfinite`	True for finite elements.	54

`isinf`	True for infinite elements.	54
`isinteger`	True for arrays of integer data type.	54
`isletter`	True for letters of the alphabet.	54, 693
`islogical`	True for logical array.	27
`islogical`	True for logical array.	54
`ismember`	True for set member.	54
`isnan`	True for Not-a-Number.	54
`isnumeric`	True for numeric arrays.	54
`ispc`	True for the PC (Windows) version of MATLAB.	728
`isprime`	True for prime numbers.	54
`isreal`	True for real array.	54
`isscalar`	True if array is a scalar.	54
`isspace`	True for white space characters.	54
`isspace`	True for white space characters.	693
`issparse`	True for sparse matrix.	54
`isstruct`	True for structures.	54
`isvector`	True if array is a vector.	54

J

`j`	Imaginary unit.	55
`jet`	Variant of HSV.	324

L

`length`	Length of vector.	59
`lines`	Color map with the line colors.	324
`linspace`	Linearly spaced vector.	34
`log`	Natural logarithm.	84
`log10`	Common (base 10) logarithm.	84
`log2`	Base 2 logarithm and dissect floating point number.	84
`logical`	Convert numeric values to logical.	27
`lookfor`	Search all M-files for keyword.	46
`lower`	Convert string to lowercase.	201, 693

M

`magic`	Magic square.	44
`makecounter`	Used by NESTEDDEMO.	141
`mat2str`	Convert a 2-D matrix to a string in MATLAB syntax.	699
`max`	Largest component.	48
`mean`	Average or mean value.	76, 517
`median`	Median value.	126
`mesh`	3-D mesh surface.	190
`meshgrid`	X and Y arrays for 3-D plots.	69
`mfilename`	Name of currently executing M-file.	730
`min`	Smallest component.	48
`movie2avi`	Create AVI movie from MATLAB movie.	475

N

NaN	Not-a-Number.	55
nargchk	Validate number of input arguments.	88
nargin	Number of function input arguments.	87
nargout	Number of function output arguments.	87
ndims	Number of dimensions.	42
nextpow2	Next higher power of 2.	175
norm	Matrix or vector norm.	675
num2str	Convert numbers to a string.	693
numel	Number of elements in an array or subscripted array expression.	59

O

ones	Ones array.	44

P

permute	Permute array dimensions.	677
persistent	Define persistent variable.	507
pi	3.1415926535897....	55
pink	Pastel shades of pink color map.	324
plot	Linear plot.	41, 98
pol2cart	Transform polar to Cartesian coordinates.	621
pow2	Base 2 power and scale floating point number.	438
print	Print figure or model. Save to disk as image or M-file.	25
prism	Prism color map.	324
prod	Product of elements.	119

Q

quad	Numerical integration based on quadratures.	56

R

rand	Uniformly distributed pseudorandom numbers.	44, 215
randn	Normally distributed pseudorandom numbers.	44, 215
real	Complex real part.	170
realmax	Largest positive floating point number.	55
realmin	Smallest positive normalized floating point number.	55
regexp	Match regular expression.	695
regexpi	Match regular expression, ignoring case.	696
regexprep	Replace string using regular expression.	696
rem	Remainder after division.	152
reshape	Change size.	401, 438
return	Return to invoking function.	58
rexexpi	NOT FOUND.	693
rgb2hsv	Convert red-green-blue colors to hue-saturation-value.	330
round	Round towards nearest integer.	25
rot90	Rotate matrix 90 degrees.	115

S

`set`	Set object properties.	96
`setfield`	Set structure field contents.	743
`shading`	Color shading mode.	194
`single`	Convert to single precision.	26
`size`	Size of array.	16
`sort`	Sort in ascending or descending order.	431
`sortrows`	Sort rows in ascending order.	604
`sparse`	Create sparse matrix.	42
`spline`	Cubic spline data interpolation.	352
`spring`	Shades of magenta and yellow color map.	324
`sprintf`	Write formatted data to string.	60, 693
`sscanf`	Read string under format control.	693
`stem`	Discrete sequence or "stem" plot.	96
`str2double`	Convert string to double precision value.	693
`str2num`	Convert string matrix to numeric array.	693
`strcat`	Concatenate strings.	696
`strcmp`	Compare strings.	73, 697
`strcmpi`	Compare strings ignoring case.	74, 454, 697
`strfind`	Find one string within another.	698
`strjust`	Justify character array.	698
`strmatch`	Find possible matches for string.	693
`strncmp`	Compare first N characters of strings.	697
`strncmpi`	Compare first N characters of strings ignoring case.	698
`strread`	Read formatted data from string.	73
`strread`	Read formatted data from string.	693
`strrep`	Replace string with another.	698
`strtok`	Find token in string.	699
`strvcat`	Vertically concatenate strings.	697
`sub2ind`	Linear index from multiple subscripts.	40
`subplot`	Create axes in tiled positions.	384
`sum`	Sum of elements.	37
`summer`	Shades of green and yellow color map.	324
`surf`	3-D colored surface.	193
`switch`	Switch among several cases based on expression.	62

T

`text`	Text annotation.	96
`tic`	Start a stopwatch timer.	65
`title`	Graph title.	96
`toc`	Read the stopwatch timer.	65
`transpose`	Transpose.	33
`true`	True array.	44, 587
`try`	Begin TRY block.	58

U

`uicontrol`	Create user interface control.	731
`uint16`	Convert to unsigned 16-bit integer.	26

`uint32`	Convert to unsigned 32-bit integer.	26
`uint8`	Convert to unsigned 8-bit integer.	26
`uiresume`	Resume execution of blocked M-file.	737
`uiwait`	Block execution and wait for resume.	737
`unique`	Set unique.	604
`upper`	Convert string to uppercase.	201, 693

V

`varargin`	Variable length input argument list.	88
`varargout`	Variable length output argument list.	88
`ver`	MATLAB, Simulink and toolbox version information.	55
`version`	MATLAB version number.	55
`view`	3-D graph viewpoint specification.	191

W

`waitbar`	Display wait bar.	151
`while`	Repeat statements an indefinite number of times.	61
`white`	All white color map.	324
`whitebg`	Change axes background color.	322
`whos`	List current variables, long form.	17
`winter`	Shades of blue and green color map.	324

X

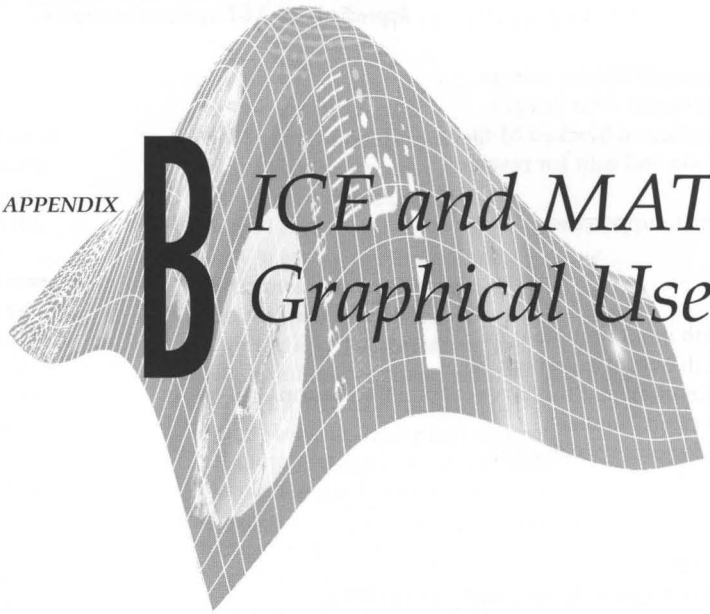
`xlabel`	X-axis label.	96
`xlim`	X limits.	98
`xor`	Logical EXCLUSIVE OR.	53

Y

`ylabel`	Y-axis label.	96
`ylim`	Y limits.	98

Z

`zeros`	Zeros array.	44



APPENDIX B

ICE and MATLAB Graphical User Interfaces

Preview

In this appendix we develop the `ice` *interactive color editing* (ICE) function introduced in Chapter 7. The discussion assumes familiarity on the part of the reader with the material in Section 7.4. Section 7.4 provides many examples of using `ice` in both pseudo- and full-color image processing (Examples 7.5 through 7.9) and describes the `ice` calling syntax, input parameters, and graphical interface elements (they are summarized in Tables 7.7 through 7.9). The power of `ice` is its ability to let users generate color transformation curves interactively and graphically, while displaying the impact of the generated transformations on images in real or near real time.

B.1 Creating ICE's Graphical User Interface

MATLAB's *Graphical User Interface Development Environment* (GUIDE) provides a rich set of tools for incorporating *graphical user interfaces* (GUIs) in M-functions. Using GUIDE, the processes of (1) laying out a GUI (i.e., its buttons, pop-up menus, etc.) and (2) programming the operation of the GUI are divided conveniently into two easily managed and relatively independent tasks. The resulting graphical M-function is composed of two identically named (ignoring extensions) files:

1. A file with extension `.fig`, called a *FIG-file*, that contains a complete graphical description of all the function's GUI objects or elements and their spatial arrangement. A FIG-file contains binary data that does not need to be parsed when the associated GUI-based M-function is executed. The FIG-file for ICE (`ice.fig`) is described later in this section.
2. A file with extension `.m`, called a *GUI M-file*, which contains the code that controls the GUI operation. This file includes functions that are called

when the GUI is launched and exited, and *callback functions* that are executed when a user interacts with GUI objects—for example, when a button is pushed. The GUI M-file for ICE (`ice.m`) is described in the next section.

To launch GUIDE from the MATLAB command window, type

```
guide filename
```



where `filename` is the name of an existing FIG-file on the current path. If `filename` is omitted, GUIDE opens a new (i.e., blank) window.

Figure B.1 shows the GUIDE *Layout Editor* (launched by entering `guide ice` at the MATLAB >> prompt) for the Interactive Color Editor (ICE) layout. The Layout Editor is used to select, place, size, align, and manipulate graphic objects on a mock-up of the user interface under development. The buttons on its left side form a *Component Palette* containing the GUI objects that are supported—*Push Buttons*, *Sliders*, *Radio Buttons*, *Checkboxes*, *Edit Texts*, *Static Texts*, *Popup Menus*, *Listboxes*, *Toggle Buttons*, *Tables*, *Axes*, *Panels*, *Button Groups*, and *ActiveX Controls*. Each object is similar in behavior to its standard Windows' counterpart. And any combination of objects can be added to the figure object in the layout area on the right side of the Layout Editor. Note that the ICE GUI includes checkboxes (`Smooth`, `Clamp Ends`, `Show PDF`, `Show`

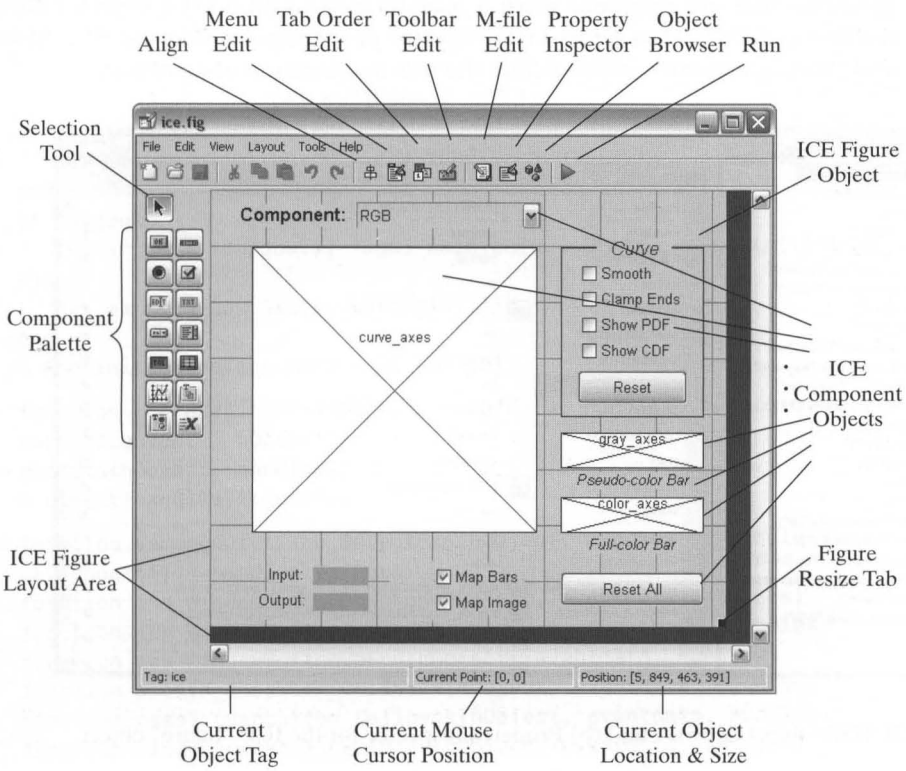
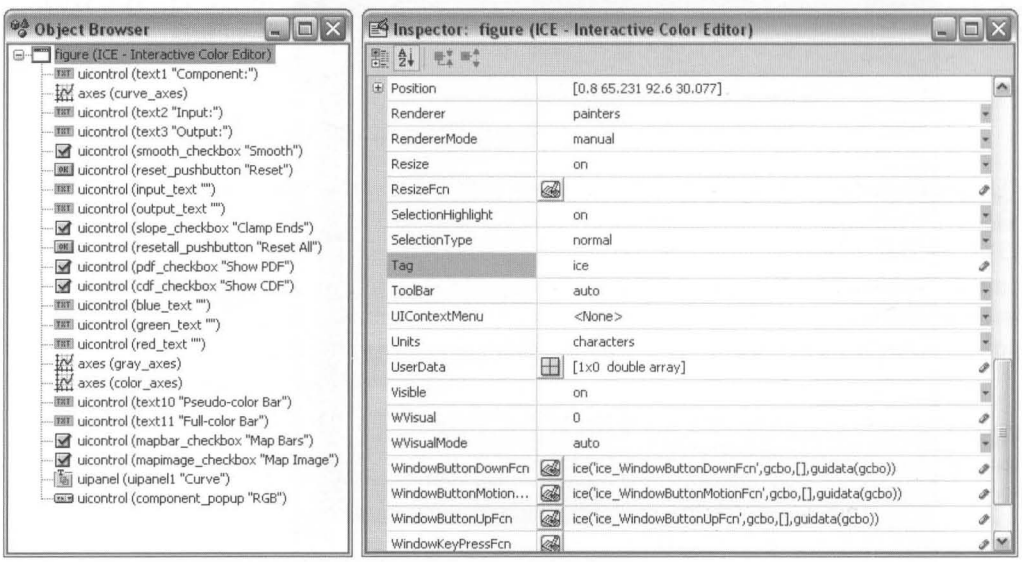


FIGURE B.1
The GUIDE Layout Editor mockup of the ICE GUI.

`CDF`, `Map Bars`, and `Map Image`), static text ("Component:", "Input:", ...), a panel outlining the curve controls, two push buttons (`Reset` and `Reset All`), a popup menu for selecting a color transformation curve, and three `axes` objects for displaying the selected curve (with associated control points) and its effect on both a gray-scale wedge and hue wedge. A hierarchical list of the elements comprising ICE (obtained by clicking the *Object Browser* button in the task bar at the top of the Layout Editor) is shown in Fig. B.2(a). Note that each element has been given a unique name or tag. For example, the axes object for curve display (at the top of the list) is assigned the identifier `curve_axes` [the identifier is the first entry after the open parenthesis in Fig. B.2(a)].

Tags are one of several *properties* that are common to all GUI objects. A scrollable list of the properties characterizing a specific object can be obtained by selecting the object [in the Object Browser list of Fig. B.2(a) or layout area of Fig. B.1 using the *Selection Tool*] and clicking the *Property Inspector* button on the Layout Editor's task bar. Figure B.2(b) shows the list that is generated when the `figure` object of Fig. B.2(a) is selected. Note that the `figure` object's `Tag` property [highlighted in Fig. B.2(b)] is `ice`. This property is important because GUIDE uses it to automatically generate `figure` callback function names. Thus, for example, the `WindowButtonDownFcn` property at the bottom of the scrollable Property Inspector window, which is executed when a mouse button is pressed over the figure window, is assigned the name `ice_WindowButtonDownFcn`. Recall that callback functions are merely M-functions that are executed when a user interacts with a GUI object. Other notable (and common to all GUI objects) properties include the `Position` and `Units` properties, which define the size and location of an object.

The GUIDE generated `figure` object is a container for all other objects in the interface.



a b

FIGURE B.2 The (a) GUIDE Object Browser and (b) Property Inspector for the ICE "figure" object.

Finally, we note that some properties are unique to particular objects. A pushbutton object, for example, has a `Callback` property that defines the function that is executed when the button is pressed and a `String` property that determines the button's label. The `Callback` property of the ICE `Reset` button is `reset_pushbutton_Callback` [note the incorporation of its `Tag` property from Fig. B.2(a) in the callback function name]; its `String` property is "Reset". Note, however, that the `Reset` pushbutton does not have a `WindowButtonMotionFcn` property; it is specific to "figure" objects.

B.2 Programming the ICE Interface

When the ICE FIG-file of the previous section is first saved or the GUI is first run (e.g., by clicking the *Run* button on the Layout Editor's task bar), GUIDE generates a starting *GUI M-file* called `ice.m`. This file, which can be modified using a standard text editor or MATLAB's M-file editor, determines how the interface responds to user actions. The automatically generated GUI M-file for ICE is as follows:

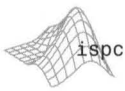
To enable M-file generation, select **Tools** and **GUI Options ...** and check the "Generate FIG-file and M-file" option.

ice

GUIDE generated starting M-file.

```
function varargout = ice(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',        mfilename, ...
                   'gui_Singleton',   gui_Singleton, ...
                   'gui_OpeningFcn',  @ice_OpeningFcn, ...
                   'gui_OutputFcn',   @ice_OutputFcn, ...
                   'gui_LayoutFcn',   [], ...
                   'gui_Callback',    []);
if nargin & ischar(varargin{1})
   gui_State.gui_Callback = str2func(varargin{1});
end
if nargout
   [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
   gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

function ice_OpeningFcn(hObject, eventdata, handles, varargin)
handles.output = hObject;
guidata(hObject, handles);
% uiwait(handles.figure1);

function varargout = ice_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;
function ice_WindowButtonDownFcn(hObject, eventdata, handles)
function ice_WindowButtonMotionFcn(hObject, eventdata, handles)
function ice_WindowButtonUpFcn(hObject, eventdata, handles)
function smooth_checkbox_Callback(hObject, eventdata, handles)
function reset_pushbutton_Callback(hObject, eventdata, handles)
function slope_checkbox_Callback(hObject, eventdata, handles)
```

```
function resetall_pushbutton_Callback(hObject, eventdata, handles)
function pdf_checkbox_Callback(hObject, eventdata, handles)
function cdf_checkbox_Callback(hObject, eventdata, handles)
function mapbar_checkbox_Callback(hObject, eventdata, handles)
function mapimage_checkbox_Callback(hObject, eventdata, handles)
function component_popup_Callback(hObject, eventdata, handles)
function component_popup_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), ...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```



Returns 1 for PC (Windows) versions of MATLAB and 0 otherwise.

This automatically generated file is a useful starting point or prototype for the development of the fully functional `ice` interface. (Note that we have stripped the file of many GUIDE-generated comments to save space.) In the sections that follow, we break this code into four basic sections: (1) the initialization code between the two "`DO NOT EDIT`" comment lines, (2) the figure opening and output functions (`ice_OpeningFcn` and `ice_OutputFcn`), (3) the figure callback functions (i.e., the `ice_WindowButtonDownFcn`, `ice_WindowButtonMotionFcn`, and `ice_WindowButtonUpFcn` functions), and (4) the object callback functions (e.g., `reset_pushbutton_Callback`). When considering each section, completely developed versions of the `ice` functions contained in the section are given, and the discussion is focused on features of general interest to most GUI M-file developers. The code introduced in each section will not be consolidated (for the sake of brevity) into a single comprehensive listing of `ice.m`. It is introduced in a piecemeal manner.

The operation of `ice` was described in Section 7.4. It is also summarized in the following Help text block from the fully developed `ice.m` M-function:

ice

Help text block of the final version.

```
%ICE Interactive Color Editor.
%
%   OUT = ICE('Property Name', 'Property Value', ...) transforms an
%   image's color components based on interactively specified mapping
%   functions. Inputs are Property Name/Property Value pairs:
%
%     Name            Value
%     ------------    ------------------------------------------------
%     'image'         An RGB or monochrome input image to be
%                     transformed by interactively specified
%                     mappings.
%     'space'         The color space of the components to be
%                     modified. Possible values are 'rgb', 'cmy',
%                     'hsi', 'hsv', 'ntsc' (or 'yiq'), 'ycbcr'. When
%                     omitted, the RGB color space is assumed.
%     'wait'          If 'on' (the default), OUT is the mapped input
%                     image and ICE returns to the calling function
%                     or workspace when closed. If 'off', OUT is the
%                     handle of the mapped input image and ICE
%                     returns immediately.
```

```
%
%   EXAMPLES:
%     ice OR ice('wait', 'off')            % Demo user interface
%     ice('image', f)                      % Map RGB or mono image
%     ice('image', f, 'space', 'hsv')      % Map HSV of RGB image
%     g = ice('image', f)                  % Return mapped image
%     g = ice('image', f, 'wait', 'off');  % Return its handle
%
%   ICE displays one popup menu selectable mapping function at a
%   time. Each image component is mapped by a dedicated curve (e.g.,
%   R, G, or B) and then by an all-component curve (e.g., RGB). Each
%   curve's control points are depicted as circles that can be moved,
%   added, or deleted with a two- or three-button mouse:
%
%     Mouse Button    Editing Operation
%     ------------    ------------------------------------------------
%     Left            Move control point by pressing and dragging.
%     Middle          Add and position a control point by pressing
%                     and dragging. (Optionally Shift-Left)
%     Right           Delete a control point. (Optionally
%                     Control-Left)
%
%   Checkboxes determine how mapping functions are computed, whether
%   the input image and reference pseudo- and full-color bars are
%   mapped, and the displayed reference curve information (e.g.,
%   PDF):
%
%     Checkbox        Function
%     ------------    ------------------------------------------------
%     Smooth          Checked for cubic spline (smooth curve)
%                     interpolation. If unchecked, piecewise linear.
%     Clamp Ends      Checked to force the starting and ending curve
%                     slopes in cubic spline interpolation to 0. No
%                     effect on piecewise linear.
%     Show PDF        Display probability density function(s) [i.e.,
%                     histogram(s)] of the image components affected
%                     by the mapping function.
%     Show CDF        Display cumulative distributions function(s)
%                     instead of PDFs.
%                     <Note: Show PDF/CDF are mutually exclusive.>
%     Map Image       If checked, image mapping is enabled; else
%                     not.
%     Map Bars        If checked, pseudo- and full-color bar mapping
%                     is enabled; else display the unmapped bars (a
%                     gray wedge and hue wedge, respectively).
%
%   Mapping functions can be initialized via pushbuttons:
%
%     Button          Function
%     ------------    ------------------------------------------------
```

```
%     Reset          Init the currently displayed mapping function
%                    and uncheck all curve parameters.
%     Reset All      Initialize all mapping functions.
```

B.2.1 Initialization Code

The opening section of code in the starting GUI M-file (at the beginning of Section B.2) is a standard GUIDE-generated block of initialization code. Its purpose is to build and display ICE's GUI using the M-file's companion FIG-file (see Section B.1) and control access to all internal M-file functions. As the enclosing "`DO NOT EDIT`" comment lines indicate, the initialization code should not be modified. Each time `ice` is called, the initialization block builds a structure called `gui_State`, which contains information for accessing `ice` functions. For instance, named field `gui_Name` (i.e., `gui_State.gui_Name`) contains the MATLAB function `mfilename`, which returns the name of the currently executing M-file. In a similar manner, fields `gui_OpeningFcn` and `gui_OutputFcn` are loaded with the GUIDE generated names of `ice`'s opening and output functions (discussed in the next section). If an ICE GUI object is activated by the user (e.g., a button is pressed), the name of the object's callback function is added as field `gui_Callback` [the callback's name would have been passed as a string in `varargin(1)`].





After structure `gui_State` is formed, it is passed as an input argument, along with `varargin(:)`, to function `gui_mainfcn`. This MATLAB function handles GUI creation, layout, and callback dispatch. For `ice`, it builds and displays the user interface and generates all necessary calls to its opening, output, and callback functions. Since older versions of MATLAB may not include this function, GUIDE is capable of generating a stand-alone version of the normal GUI M-file (i.e., one in which the FIG-file is replaced with a MAT-file) by selecting **Export ...** from the **File** menu. In the stand-alone version, function `gui_mainfcn` and several supporting routines, including `ice_LayoutFcn` and `local_openfig`, are appended to the normally FIG-file dependent M-file. The role of `ice_LayoutFcn` is to create the ICE GUI. In the stand-alone version of `ice`, it includes the statement

To choose a compatibility mode, select **File** and **Preferences ...**, followed by *General* and *MAT-Files*, and choose a MAT-file save format.

```
h1 = figure(...
'Units', 'characters',...
'Color', [0.87843137254902 0.874509803921569 0.890196078431373],...
'Colormap', [0 0 0.5625;0 0 0.625;0 0 0.6875;0 0 0.75;...
             0 0 0.8125;0 0 0.875;0 0 0.9375;0 0 1;0 0.0625 1;...
             0 0.125 1;0 0.1875 1;0 0.25 1;0 0.3125 1;0 0.375 1;...
             0 0.4375 1;0 0.5 1;0 0.5625 1;0 0.625 1;0 0.6875 1;...
             0 0.75 1;0 0.8125 1;0 0.875 1;0 0.9375 1;0 1 1;...
             0.0625 1 1;0.125 1 0.9375;0.1875 1 0.875;...
             0.25 1 0.8125;0.3125 1 0.75;0.375 1 0.6875;...
             0.4375 1 0.625;0.5 1 0.5625;0.5625 1 0.5;...
             0.625 1 0.4375;0.6875 1 0.375;0.75 1 0.3125;...
             0.8125 1 0.25;0.875 1 0.1875;0.9375 1 0.125;...
             1 1 0.0625;1 1 0;1 0.9375 0;1 0.875 0;1 0.8125 0;...
```

```
            1 0.75 0;1 0.6875 0;1 0.625 0;1 0.5625 0;1 0.5 0;...
            1 0.4375 0;1 0.375 0;1 0.3125 0;1 0.25 0;...
            1 0.1875 0;1 0.125 0;1 0.0625 0;1 0 0;0.9375 0 0;...
            0.875 0 0;0.8125 0 0;0.75 0 0;0.6875 0 0;0.625 0 0;...
            0.5625 0 0],...
'IntegerHandle', 'off',...
'InvertHardcopy', get(0, 'defaultfigureInvertHardcopy'),...
'MenuBar', 'none',...
'Name', 'ICE - Interactive Color Editor',...
'NumberTitle', 'off',...
'PaperPosition', get(0, 'defaultfigurePaperPosition'),...
'Position', [0.8 65.2307692307693 92.6 30.0769230769231],...
'Renderer', get(0, 'defaultfigureRenderer'),...
'RendererMode', 'manual',...
'WindowButtonDownFcn', 'ice(''ice_WindowButtonDownFcn'', gcbo, [],...
                        guidata(gcbo))',...
'WindowButtonMotionFcn', 'ice(''ice_WindowButtonMotionFcn'', gcbo,...
                        [], guidata(gcbo))',...
'WindowButtonUpFcn', 'ice(''ice_WindowButtonUpFcn'', gcbo, [],...
                        guidata(gcbo))',...
'HandleVisibility', 'callback',...
'Tag', 'ice',...
'UserData',[],...
'CreateFcn', {@local_CreateFcn, blanks(0), appdata} );
```

to create the main figure window. GUI objects are then added with statements like

```
h11 = uicontrol(...
'Parent',h1,...
'Units','normalized',...
'Callback',mat{5},...
'FontSize',10,...
'ListboxTop',0,...
'Position',[0.710583153347732 0.508951406649616 0.211663066954644
0.0767263427109974],...
'String','Reset',...
'Tag','reset_pushbutton',...
'CreateFcn', {@local_CreateFcn, blanks(0), appdata} );
```



Function `uicontrol ('PropertyName1', Value1, ...)` creates a user interface control in the current window with the specified properties and returns a handle to it.

which adds the `Reset` pushbutton to the figure. Note that these statements specify explicitly properties that were defined originally using the Property Inspector of the GUIDE Layout Editor. Finally, we note that the `figure` function was introduced in Section 2.3; `uicontrol` creates a user interface control (i.e., GUI object) in the current figure window based on property name/value pairs (e.g., `'Tag'` plus `'reset_pushbutton'`) and returns a handle to it.

B.2.2 The Opening and Output Functions

The first two functions following the initialization block in the starting GUI M-file at the beginning of Section B.2 are called *opening* and *output functions*,

respectively. They contain the code that is executed just before the GUI is made visible to the user and when the GUI returns its output to the command line or calling routine. Both functions are passed arguments `hObject`, `eventdata`, and `handles`. (These arguments are also inputs to the callback functions in the next two sections.) Input `hObject` is a graphics object handle, `eventdata` is reserved for future use, and `handles` is a structure that provides handles to interface objects and any application specific or user defined data. To implement the desired functionality of the ICE interface (see the Help text), both `ice_OpeningFcn` and `ice_OutputFcn` must be expanded beyond the "barebones" versions in the starting GUI M-file. The expanded code is as follows:

`ice_OpeningFcn`

From the final M-file.

```
%--------------------------------------------------------------------%
function ice_OpeningFcn(hObject, eventdata, handles, varargin)
%  When ICE is opened, perform basic initialization (e.g., setup
%  globals, ...) before it is made visible.

% Set ICE globals to defaults.
handles.updown = 'none';          % Mouse updown state
handles.plotbox = [0 0 1 1];      % Plot area parameters in pixels
handles.set1 = [0 0; 1 1];        % Curve 1 control points
handles.set2 = [0 0; 1 1];        % Curve 2 control points
handles.set3 = [0 0; 1 1];        % Curve 3 control points
handles.set4 = [0 0; 1 1];        % Curve 4 control points
handles.curve = 'set1';           % Structure name of selected curve
handles.cindex = 1;               % Index of selected curve
handles.node = 0;                 % Index of selected control point
handles.below = 1;                % Index of node below control point
handles.above = 2;                % Index of node above control point
handles.smooth = [0; 0; 0; 0];    % Curve smoothing states
handles.slope = [0; 0; 0; 0];     % Curve end slope control states
handles.cdf = [0; 0; 0; 0];       % Curve CDF states
handles.pdf = [0; 0; 0; 0];       % Curve PDF states
handles.output = [];              % Output image handle
handles.df = [];                  % Input PDFs and CDFs
handles.colortype = 'rgb';        % Input image color space
handles.input = [];               % Input image data
handles.imagemap = 1;             % Image map enable
handles.barmap = 1;               % Bar map enable
handles.graybar = [];             % Pseudo (gray) bar image
handles.colorbar = [];            % Color (hue) bar image

% Process Property Name/Property Value input argument pairs.
wait = 'on';
if (nargin > 3)
   for i = 1:2:(nargin - 3)
      if nargin - 3 == i
         break;
      end
      switch lower(varargin{i})
```

```
      case 'image'
         if ndims(varargin{i + 1}) == 3
            handles.input = varargin{i + 1};
         elseif ndims(varargin{i + 1}) == 2
            handles.input = cat(3, varargin{i + 1}, ...
                                varargin{i + 1}, varargin{i + 1});
         end
         handles.input = double(handles.input);
         inputmax = max(handles.input(:));
         if inputmax > 255
            handles.input = handles.input / 65535;
         elseif inputmax > 1
            handles.input = handles.input / 255;
         end

      case 'space'
         handles.colortype = lower(varargin{i + 1});
         switch handles.colortype
         case 'cmy'
            list = {'CMY' 'Cyan' 'Magenta' 'Yellow'};
         case {'ntsc', 'yiq'}
            list = {'YIQ' 'Luminance' 'Hue' 'Saturation'};
            handles.colortype = 'ntsc';
         case 'ycbcr'
            list = {'YCbCr' 'Luminance' 'Blue' ...
                    'Difference'  'Red Difference'};
         case 'hsv'
            list = {'HSV' 'Hue' 'Saturation' 'Value'};
         case 'hsi'
            list = {'HSI' 'Hue' 'Saturation' 'Intensity'};
         otherwise
            list = {'RGB' 'Red' 'Green' 'Blue'};
            handles.colortype = 'rgb';
         end
         set(handles.component_popup, 'String', list);

      case 'wait'
         wait = lower(varargin{i + 1});
      end
   end
end

% Create pseudo- and full-color mapping bars (grays and hues). Store
% a color space converted 1x128x3 line of each bar for mapping.
xi = 0:1/127:1;      x = 0:1/6:1;      x = x';
y = [1 1 0 0 0 1 1; 0 1 1 1 0 0 0; 0 0 0 1 1 1 0]';
gb = repmat(xi, [1 1 3]);        cb = interp1q(x, y, xi');
cb = reshape(cb, [1 128 3]);
if ~strcmp(handles.colortype, 'rgb')
   gb = eval(['rgb2' handles.colortype '(gb)']);
   cb = eval(['rgb2' handles.colortype '(cb)']);
end
```

```
gb = round(255 * gb);     gb = max(0, gb);      gb = min(255, gb);
cb = round(255 * cb);     cb = max(0, cb);      cb = min(255, cb);
handles.graybar = gb;     handles.colorbar = cb;

% Do color space transforms, clamp to [0, 255], compute histograms
% and cumulative distribution functions, and create output figure.
if size(handles.input, 1)
   if ~strcmp(handles.colortype, 'rgb')
      handles.input = eval(['rgb2' handles.colortype ...
                            '(handles.input)']);
   end
   handles.input = round(255 * handles.input);
   handles.input = max(0, handles.input);
   handles.input = min(255, handles.input);
   for i = 1:3
      color = handles.input(:, :, i);
      df = hist(color(:), 0:255);
      handles.df = [handles.df; df / max(df(:))];
      df = df / sum(df(:));   df = cumsum(df);
      handles.df = [handles.df; df];
   end
   figure;     handles.output = gcf;
end

% Compute ICE's screen position and display image/graph.
set(0, 'Units', 'pixels');      ssz = get(0, 'Screensize');
set(handles.ice, 'Units', 'pixels');
uisz = get(handles.ice, 'Position');
if size(handles.input, 1)
   fsz = get(handles.output, 'Position');
   bc = (fsz(4) - uisz(4)) / 3;
   if bc > 0
      bc = bc + fsz(2);
   else
      bc = fsz(2) + fsz(4) - uisz(4) - 10;
   end
   lc = fsz(1) + (size(handles.input, 2) / 4) + (3 * fsz(3) / 4);
   lc = min(lc, ssz(3) - uisz(3) - 10);
   set(handles.ice, 'Position', [lc bc 463 391]);
else
   bc = round((ssz(4) - uisz(4)) / 2) - 10;
   lc = round((ssz(3) - uisz(3)) / 2) - 10;
   set(handles.ice, 'Position', [lc bc uisz(3) uisz(4)]);
end
set(handles.ice, 'Units', 'normalized');
graph(handles);     render(handles);

% Update handles and make ICE wait before exit if required.
guidata(hObject, handles);
if strcmpi(wait, 'on')
   uiwait(handles.ice);
```

```
end

%--------------------------------------------------------------------%
function varargout = ice_OutputFcn(hObject, eventdata, handles)
%  After ICE is closed, get the image data of the current figure
%  for the output. If 'handles' exists, ICE isn't closed (there was
%  no 'uiwait') so output figure handle.

if max(size(handles)) == 0
   figh = get(gcf);
   imageh = get(figh.Children);
   if max(size(imageh)) > 0
      image = get(imageh.Children);
      varargout{1} = image.CData;
   end
else
   varargout{1} = hObject;
end
```

ice_OutputFcn

From the final M-file.

Rather than examining the intricate details of these functions (see the code's comments and consult Appendix A or the index for help on specific functions), we note the following commonalities with most GUI opening and output functions:

1. The `handles` structure (as can be seen from its numerous references in the code) plays a central role in most GUI M-files. It serves two crucial functions. Since it provides handles for all the graphic objects in the interface, it can be used to access and modify object properties. For instance, the `ice` opening function uses

```
set(handles.ice, 'Units', 'pixels');
uisz = get(handles.ice, 'Position');
```

to access the size and location of the ICE GUI (in pixels). This is accomplished by setting the `Units` property of the `ice` figure, whose handle is available in `handles.ice`, to `'pixels'` and then reading the `Position` property of the figure (using the `get` function). The `get` function, which returns the value of a property associated with a graphics object, is also used to obtain the computer's display area via the `ssz = get(0, 'Screensize')` statement near the end of the opening function. Here, `0` is the handle of the computer display (i.e., root figure) and `'Screensize'` is a property containing its extent.

In addition to providing access to GUI objects, the `handles` structure is a powerful conduit for sharing application data. Note that it holds the default values for twenty-three global `ice` parameters (ranging from the mouse state in `handles.updown` to the entire input image in `handles.input`). They must survive every call to `ice` and are added to `handles` at the start of `ice_OpeningFcn`. For instance, the `handles.set1` global is created by the statement

```
handles.set1 = [0 0; 1 1]
```

where `set1` is a named field containing the control points of a color mapping function to be added to the `handles` structure and `[0 0; 1 1]` is its default value [curve endpoints (0, 0) and (1, 1)]. Before exiting a function in which `handles` is modified,



Function `guidata (H, DATA)` stores the specified data in the figure's application data. `H` is a handle that identifies the figure—it can be the figure itself, or any object contained in the figure.

```
guidata(hObject, handles)
```

must be called to store variable `handles` as the application data of the figure with handle `hObject`.

2. Like many built-in graphics functions, `ice_OpeningFcn` processes input arguments (except `hObject`, `eventdata`, and `handles`) in property name and value pairs. When there are more than three input arguments (i.e., if `nargin > 3`), a loop that skips through the input arguments in pairs [`for i = 1:2:(nargin - 3)`] is executed. For each pair of inputs, the first is used to drive the `switch` construct,

```
switch lower(varargin{i})
```

which processes the second parameter appropriately. For `case 'space'`, for instance, the statement

```
handles.colortype = lower(varargin{i + 1});
```

sets named field `colortype` to the value of the second argument of the input pair. This value is then used to setup ICE's color component popup options (i.e., the `String` property of object `component_popup`). Later, it is used to transform the components of the input image to the desired mapping space via

```
handles.input = eval(['rgb2' ...
    handles.colortype '(handles.input)']);
```

where built-in function `eval(s)` causes MATLAB to execute string `s` as an expression or statement (see Section 13.4.1 for more on function `eval`). If `handles.input` is `'hsv'`, for example, `eval` argument `['rgb2' 'hsv' '(handles.input)']` becomes the concatenated string `'rgb2hsv(handles.input)'`, which is executed as a standard MATLAB expression that transforms the RGB components of the input image to the HSV color space (see Section 7.2.3).

3. The statement

```
% uiwait(handles.figure1);
```

in the starting GUI M-file is converted into the conditional statement

```
if strcmpi(wait, 'on') uiwait(handles.ice); end
```

in the final version of `ice_OpeningFcn`. In general,




```
uiwait(fig)
```

blocks execution of a MATLAB code stream until either a `uiresume` is executed or figure `fig` is destroyed (i.e., closed). [With no input arguments, `uiwait` is the same as `uiwait(gcf)` where MATLAB function `gcf` returns the handle of the current figure]. When `ice` is not expected to return a mapped version of an input image, but return immediately (i.e., before the ICE GUI is closed), an input property name/value pair of `'wait'/'off'` must be included in the call. Otherwise, ICE will not return to the calling routine or command line until it is closed—that is, until the user is finished interacting with the interface (and color mapping functions). In this situation, function `ice_OutputFcn` can not obtain the mapped image data from the `handles` structure, because it does not exist after the GUI is closed. As can be seen in the final version of the function, ICE extracts the image data from the `CData` property of the surviving mapped image output figure. If a mapped output image is not to be returned by `ice`, the `uiwait` statement in `ice_OpeningFcn` is not executed, `ice_OutputFcn` is called immediately after the opening function (long before the GUI is closed), and the handle of the mapped image output figure is returned to the calling routine or command line.

Finally, we note that several internal functions are invoked by `ice_OpeningFcn`. These—and all other `ice` internal functions—are listed next. Note that they provide additional examples of the usefulness of the `handles` structure in MATLAB GUIs. For instance, the

```
nodes = getfield(handles, handles.curve)
```

and

```
nodes = getfield(handles, ['set' num2str(i)])
```

statements in internal functions `graph` and `render`, respectively, are used to access the interactively defined control points of ICE's various color mapping curves. In its standard form,




```
F = getfield(S,'field')
```

returns to `F` the contents of named field `'field'` from structure `S`.

```
%-------------------------------------------------------------------%
function graph(handles)
%  Interpolate and plot mapping functions and optional reference
%  PDF(s) or CDF(s).

nodes = getfield(handles, handles.curve);
```

ice
Internal Functions

```
c = handles.cindex;      dfx = 0:1/255:1;
colors = ['k' 'r' 'g' 'b'];

% For piecewise linear interpolation, plot a map, map + PDF/CDF, or
% map + 3 PDFs/CDFs.
if ~handles.smooth(handles.cindex)
   if (~handles.pdf(c) && ~handles.cdf(c)) || ...
         (size(handles.df, 2) == 0)
      plot(nodes(:, 1), nodes(:, 2), 'b-', ...
           nodes(:, 1),  nodes(:, 2), 'ko', ...
           'Parent', handles.curve_axes);
   elseif c > 1
      i = 2 * c - 2 - handles.pdf(c);
      plot(dfx, handles.df(i, :), [colors(c) '-'], ...
           nodes(:, 1), nodes(:, 2), 'k-', ...
           nodes(:, 1), nodes(:, 2), 'ko', ...
           'Parent', handles.curve_axes);
   elseif c == 1
      i = handles.cdf(c);
      plot(dfx, handles.df(i + 1, :), 'r-', ...
           dfx, handles.df(i + 3, :), 'g-',  ...
           dfx, handles.df(i + 5, :), 'b-', ...
           nodes(:, 1), nodes(:, 2), 'k-',  ...
           nodes(:, 1), nodes(:, 2), 'ko', ...
           'Parent', handles.curve_axes);
   end

% Do the same for smooth (cubic spline) interpolations.
else
   x = 0:0.01:1;
   if ~handles.slope(handles.cindex)
      y = spline(nodes(:, 1), nodes(:, 2), x);
   else
      y = spline(nodes(:, 1), [0; nodes(:, 2); 0], x);
   end
   i = find(y > 1);        y(i) = 1;
   i = find(y < 0);        y(i) = 0;

   if (~handles.pdf(c) && ~handles.cdf(c)) || ...
         (size(handles.df, 2) == 0)
      plot(nodes(:, 1), nodes(:, 2), 'ko',  x, y, 'b-', ...
           'Parent', handles.curve_axes);
   elseif c > 1
      i = 2 * c - 2 - handles.pdf(c);
      plot(dfx, handles.df(i, :), [colors(c) '-'], ...
           nodes(:, 1), nodes(:, 2), 'ko', x, y, 'k-', ...
           'Parent', handles.curve_axes);
   elseif c == 1
      i = handles.cdf(c);
      plot(dfx, handles.df(i + 1, :), 'r-', ...
           dfx, handles.df(i + 3, :), 'g-', ...
           dfx, handles.df(i + 5, :), 'b-', ...
```

```
            nodes(:, 1), nodes(:, 2), 'ko', x, y, 'k-', ...
            'Parent', handles.curve_axes);
   end
end

% Put legend if more than two curves are shown.
s = handles.colortype;
if strcmp(s, 'ntsc')
    s = 'yiq';
end
if (c == 1) && (handles.pdf(c) || handles.cdf(c))
   s1 = ['-- ' upper(s(1))];
   if length(s) == 3
      s2 = ['-- ' upper(s(2))];       s3 = ['-- ' upper(s(3))];
   else
      s2 = ['-- ' upper(s(2)) s(3)];  s3 = ['-- ' upper(s(4)) s(5)];
   end
else
   s1 = '';    s2 = '';    s3 = '';
end
set(handles.red_text, 'String', s1);
set(handles.green_text, 'String', s2);
set(handles.blue_text, 'String', s3);

%--------------------------------------------------------------------%
function [inplot, x, y] = cursor(h, handles)
%  Translate the mouse position to a coordinate with respect to
%  the current plot area, check for the mouse in the area and if so
%  save the location and write the coordinates below the plot.

set(h, 'Units', 'pixels');
p = get(h, 'CurrentPoint');
x = (p(1, 1) - handles.plotbox(1)) / handles.plotbox(3);
y = (p(1, 2) - handles.plotbox(2)) / handles.plotbox(4);
if x > 1.05 || x < -0.05 || y > 1.05 || y < -0.05
   inplot = 0;
else
   x = min(x, 1);      x = max(x, 0);
   y = min(y, 1);      y = max(y, 0);
   nodes = getfield(handles, handles.curve);
   x = round(256 * x) / 256;
   inplot = 1;
   set(handles.input_text, 'String', num2str(x, 3));
   set(handles.output_text, 'String', num2str(y, 3));
end
set(h, 'Units', 'normalized');

%--------------------------------------------------------------------%
function y = render(handles)
%  Map the input image and bar components and convert them to RGB
%  (if needed) and display.

set(handles.ice, 'Interruptible', 'off');
```

```
set(handles.ice, 'Pointer', 'watch');
ygb = handles.graybar;      ycb = handles.colorbar;
yi = handles.input;         mapon = handles.barmap;
imageon = handles.imagemap & size(handles.input, 1);

for i = 2:4
   nodes = getfield(handles, ['set' num2str(i)]);
   t = lut(nodes, handles.smooth(i), handles.slope(i));
   if imageon
      yi(:, :, i - 1) = t(yi(:, :, i - 1) + 1);
   end
   if mapon
      ygb(:, :, i - 1) = t(ygb(:, :, i - 1) + 1);
      ycb(:, :, i - 1) = t(ycb(:, :, i - 1) + 1);
   end
end
t = lut(handles.set1, handles.smooth(1), handles.slope(1));
if imageon
   yi = t(yi + 1);
end
if mapon
   ygb = t(ygb + 1);     ycb = t(ycb + 1);
end

if ~strcmp(handles.colortype, 'rgb')
   if size(handles.input, 1)
      yi = yi / 255;
      yi = eval([handles.colortype '2rgb(yi)']);
      yi = uint8(255 * yi);
   end
   ygb = ygb / 255;        ycb = ycb / 255;
   ygb = eval([handles.colortype '2rgb(ygb)']);
   ycb = eval([handles.colortype '2rgb(ycb)']);
   ygb = uint8(255 * ygb);     ycb = uint8(255 * ycb);
else
   yi = uint8(yi);   ygb = uint8(ygb);     ycb = uint8(ycb);
end

if size(handles.input, 1)
   figure(handles.output);    imshow(yi);
end
ygb = repmat(ygb, [32 1 1]);    ycb = repmat(ycb, [32 1 1]);
axes(handles.gray_axes);        imshow(ygb);
axes(handles.color_axes);       imshow(ycb);
figure(handles.ice);
set(handles.ice, 'Pointer', 'arrow');
set(handles.ice, 'Interruptible', 'on');

%--------------------------------------------------------------------%
function t = lut(nodes, smooth, slope)
%  Create a 256 element mapping function from a set of control
%  points. The output values are integers in the interval [0, 255].
```

```
%  Use piecewise linear or cubic spline with or without zero end
%  slope interpolation.

t = 255 * nodes;     i = 0:255;
if ~smooth
   t = [t; 256 256];   t = interp1q(t(:, 1), t(:, 2), i');
else
   if ~slope
      t = spline(t(:, 1), t(:, 2), i);
   else
      t = spline(t(:, 1), [0; t(:, 2); 0], i);
   end
end
t = round(t);        t = max(0, t);          t = min(255, t);

%--------------------------------------------------------------------%
function out = spreadout(in)
%  Make all x values unique.

% Scan forward for non-unique x's and bump the higher indexed x--
% but don't exceed 1. Scan the entire range.
nudge = 1 / 256;
for i = 2:size(in, 1) - 1
   if in(i, 1) <= in(i - 1, 1)
      in(i, 1) = min(in(i - 1, 1) + nudge, 1);
   end
end

% Scan in reverse for non-unique x's and decrease the lower indexed
% x -- but don't go below 0. Stop on the first non-unique pair.
if in(end, 1) == in(end - 1, 1)
   for i = size(in, 1):-1:2
      if in(i, 1) <= in(i - 1, 1)
         in(i - 1, 1) = max(in(i, 1) - nudge, 0);
      else
         break;
      end
   end
end

% If the first two x's are now the same, init the curve.
if in(1, 1) == in(2, 1)
   in = [0 0; 1 1];
end
out = in;

%--------------------------------------------------------------------%
function g = rgb2cmy(f)
%   Convert RGB to CMY using IPT's imcomplement.

g = imcomplement(f);

%--------------------------------------------------------------------%
function g = cmy2rgb(f)
```

```
%   Convert CMY to RGB using IPT's imcomplement.
g = imcomplement(f);
```

B.2.3 Figure Callback Functions

The three functions immediately following the ICE opening and closing functions in the starting GUI M-file at the beginning of Section B.2 are *figure callbacks* `ice_WindowButtonDownFcn`, `ice_WindowButtonMotionFcn`, and `ice_WindowButtonUpFcn`. In the automatically generated M-file, they are *function stubs*—that is, MATLAB `function` definition statements without supporting code. Fully developed versions of the three functions, whose joint task is to process mouse events (clicks and drags of mapping function control points on ICE's `curve_axes` object), are as follows:

ice
Figure Callbacks

```
%-------------------------------------------------------------------%
function ice_WindowButtonDownFcn(hObject, eventdata, handles)
%  Start mapping function control point editing. Do move, add, or
%  delete for left, middle, and right button mouse clicks ('normal',
%  'extend', and 'alt' cases) over plot area.

set(handles.curve_axes, 'Units', 'pixels');
handles.plotbox = get(handles.curve_axes, 'Position');
set(handles.curve_axes, 'Units', 'normalized');
[inplot, x, y] = cursor(hObject, handles);
if inplot
   nodes = getfield(handles, handles.curve);
   i = find(x >= nodes(:, 1));      below = max(i);
   above = min(below + 1, size(nodes, 1));
   if (x - nodes(below, 1)) > (nodes(above, 1) - x)
      node = above;
   else
      node = below;
   end
   deletednode = 0;

   switch get(hObject, 'SelectionType')
   case 'normal'
      if node == above
         above = min(above + 1, size(nodes, 1));
      elseif node == below
         below = max(below - 1, 1);
      end
      if node == size(nodes, 1)
         below = above;
      elseif node == 1
         above = below;
      end
      if x > nodes(above, 1)
         x = nodes(above, 1);
      elseif x < nodes(below, 1)
```

```
        x = nodes(below, 1);
      end
      handles.node = node;      handles.updown = 'down';
      handles.below = below;    handles.above = above;
      nodes(node, :) = [x y];
   case 'extend'
      if ~any(nodes(:, 1) == x)
         nodes = [nodes(1:below, :); [x y]; nodes(above:end, :)];
         handles.node = above;   handles.updown = 'down';
         handles.below = below;  handles.above = above + 1;
      end
   case 'alt'
      if (node ~= 1) && (node ~= size(nodes, 1))
         nodes(node, :) = [];   deletednode = 1;
      end
      handles.node = 0;
      set(handles.input_text, 'String', '');
      set(handles.output_text, 'String', '');
   end

   handles = setfield(handles, handles.curve, nodes);
   guidata(hObject, handles);
   graph(handles);
   if deletednode
      render(handles);
   end
end

%--------------------------------------------------------------------%
function ice_WindowButtonMotionFcn(hObject, eventdata, handles)
%  Do nothing unless a mouse 'down' event has occurred. If it has,
%  modify control point and make new mapping function.

if ~strcmpi(handles.updown, 'down')
   return;
end
[inplot, x, y] = cursor(hObject, handles);
if inplot
   nodes = getfield(handles, handles.curve);
   nudge = handles.smooth(handles.cindex) / 256;
   if (handles.node ~= 1) && (handles.node ~= size(nodes, 1))
      if x >= nodes(handles.above, 1)
         x = nodes(handles.above, 1) - nudge;
      elseif x <= nodes(handles.below, 1)
         x = nodes(handles.below, 1) + nudge;
      end
   else
      if x > nodes(handles.above, 1)
         x = nodes(handles.above, 1);
      elseif x < nodes(handles.below, 1)
         x = nodes(handles.below, 1);
      end
   end
```



Functions `S = setfield(S, 'field', V)` sets the contents of the specified field to value `V`. The changed structure is returned.

```
   nodes(handles.node, :) = [x y];
   handles = setfield(handles, handles.curve, nodes);
   guidata(hObject, handles);
   graph(handles);
end

%--------------------------------------------------------------------%
function ice_WindowButtonUpFcn(hObject, eventdata, handles)
%  Terminate ongoing control point move or add operation. Clear
%  coordinate text below plot and update display.

update = strcmpi(handles.updown, 'down');
handles.updown = 'up';      handles.node = 0;
guidata(hObject, handles);
if update
   set(handles.input_text, 'String', '');
   set(handles.output_text, 'String', '');
   render(handles);
end
```

In general, figure callbacks are launched in response to interactions with a figure object or window—not an active `uicontrol` object. More specifically,

- The `WindowButtonDownFcn` is executed when a user clicks a mouse button with the cursor in a figure but not over an enabled `uicontrol` (e.g., a pushbutton or popup menu).
- The `WindowButtonMotionFcn` is executed when a user moves a depressed mouse button within a figure window.
- The `WindowButtonUpFcn` is executed when a user releases a mouse button, after having pressed the mouse button within a figure but not over an enabled `uicontrol`.

The purpose and behavior of `ice`'s figure callbacks are documented (via comments) in the code. We make the following general observations about the final implementations:

1. Because the `ice_WindowButtonDownFcn` is called on all mouse button clicks in the `ice` figure (except over an active graphic object), the first job of the callback function is to see if the cursor is within `ice`'s plot area (i.e., the extent of the `curve_axes` object). If the cursor is outside this area, the mouse should be ignored. The test for this is performed by internal function `cursor`, whose listing was provided in the previous section. In cursor, the statement

   ```
   p = get(h, 'CurrentPoint');
   ```

   returns the current cursor coordinates. Variable `h` is passed from `ice_WindowButtonDownFcn` and originates as input argument `hObject`. In all figure callbacks, `hObject` is the handle of the figure requesting service.

Property `'CurrentPoint'` contains the position of the cursor relative to the figure as a two-element row vector [*x y*].

2. Since `ice` is designed to work with two- and three-button mice, `ice_WindowButtonDownFcn` must determine which mouse button causes each callback. As can be seen in the code, this is done with a `switch` construct using the figure's `'SelectionType'` property. Cases `'normal'`, `'extent'`, and `'alt'` correspond to the left, middle, and right button clicks on three-button mice (or the left, shift-left, and control-left clicks of two-button mice), respectively, and are used to trigger the add control point, move control point, and delete control point operations.

3. The displayed ICE mapping function is updated (via internal function `graph`) each time a control point is modified, but the output figure, whose handle is stored in `handles.output`, is updated on *mouse button releases only*. This is because the computation of the output image, which is performed by internal function `render`, can be time-consuming. It involves mapping separately the input image's three color components, remapping each by the "all-component" curve, and converting the mapped components to the RGB color space for display. Note that without adequate precautions, the mapping function's control points could be modified inadvertently during this lengthy output mapping process.

   To prevent this, `ice` controls the interruptibility of its various callbacks. All MATLAB graphics objects have an `Interruptible` property that determines whether their callbacks can be interrupted. The default value of every object's `'Interruptible'` property is `'on'`, which means that object callbacks can be interrupted. If switched to `'off'`, callbacks that occur during the execution of the *now* noninterruptible callback are either ignored (i.e., cancelled) or placed in an *event queue* for later processing. The disposition of the interrupting callback is determined by the `'BusyAction'` property of the object being interrupted. If `'BusyAction'` is `'cancel'`, the callback is discarded; if `'queue'`, the callback is processed after the noninterruptible callback finishes.

   The `ice_WindowButtonUpFcn` function uses the mechanism just described to suspend temporarily (i.e., during output image computations) the user's ability to manipulate mapping function control points. The sequence

```
set(handles.ice, 'Interruptible', 'off');
set(handles.ice, 'Pointer', 'watch');

set(handles.ice, 'Pointer', 'arrow');
set(handles.ice, 'Interruptible', 'on');
```

   in internal function `render` sets the `ice` figure window's `'Interruptible'` property to `'off'` during the mapping of the output image and pseudo- and full-color bars. This prevents users from modifying mapping function control points while a mapping is being performed. Note also that the figure's

'Pointer' property is set to 'watch' to indicate visually that ice is busy and reset to 'arrow' when the output computation is completed.

B.2.4 Object Callback Functions

The final fourteen lines (i.e., ten functions) of the starting GUI M-file at the beginning of Section B.2 are *object callback* function stubs. Like the automatically generated figure callbacks of the previous section, they are initially void of code. Fully developed versions of the functions follow. Note that each function processes user interaction with a different ice uicontrol object (pushbutton, etc.) and is named by concatenating its Tag property with string '_Callback'. For example, the callback function responsible for handling the selection of the displayed mapping function is named the component_popup_Callback. It is called when the user activates (i.e., clicks on) the popup selector. Note also that input argument hObject is the handle of the popup graphics object—not the handle of the ice figure (as in the figure callbacks of the previous section). ICE's object callbacks involve minimal code and are self-documenting. Because ice does not use context-sensitive (i.e., right-click initiated) menus, function stub component_popup_CreateFcn is left in its intially void state. It is a callback routine that is executed during object creation.

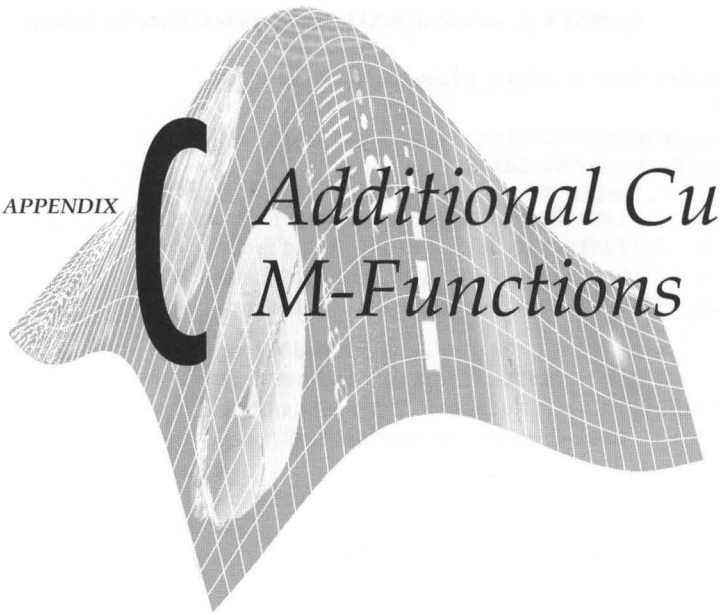
ice
Object Callbacks

```
%--------------------------------------------------------------------%
function smooth_checkbox_Callback(hObject, eventdata, handles)
%  Accept smoothing parameter for currently selected color
%  component and redraw mapping function.

if get(hObject, 'Value')
   handles.smooth(handles.cindex) = 1;
   nodes = getfield(handles, handles.curve);
   nodes = spreadout(nodes);
   handles = setfield(handles, handles.curve, nodes);
else
   handles.smooth(handles.cindex) = 0;
end
guidata(hObject, handles);
set(handles.ice, 'Pointer', 'watch');
graph(handles);      render(handles);
set(handles.ice, 'Pointer', 'arrow');

%--------------------------------------------------------------------%
function reset_pushbutton_Callback(hObject, eventdata, handles)
%  Init all display parameters for currently selected color
%  component, make map 1:1, and redraw it.

handles = setfield(handles, handles.curve, [0 0; 1 1]);
c = handles.cindex;
handles.smooth(c) = 0;  set(handles.smooth_checkbox, 'Value', 0);
handles.slope(c) = 0;   set(handles.slope_checkbox, 'Value', 0);
handles.pdf(c) = 0;     set(handles.pdf_checkbox, 'Value', 0);
```

```
handles.cdf(c) = 0;     set(handles.cdf_checkbox, 'Value', 0);
guidata(hObject, handles);
set(handles.ice, 'Pointer', 'watch');
graph(handles);      render(handles);
set(handles.ice, 'Pointer', 'arrow');

%--------------------------------------------------------------------%
function slope_checkbox_Callback(hObject, eventdata, handles)
%  Accept slope clamp for currently selected color component and
%  draw function if smoothing is on.

if get(hObject, 'Value')
   handles.slope(handles.cindex) = 1;
else
   handles.slope(handles.cindex) = 0;
end
guidata(hObject, handles);
if handles.smooth(handles.cindex)
   set(handles.ice, 'Pointer', 'watch');
   graph(handles);      render(handles);
   set(handles.ice, 'Pointer', 'arrow');
end

%--------------------------------------------------------------------%
function resetall_pushbutton_Callback(hObject, eventdata, handles)
%  Init display parameters for color components, make all maps 1:1,
%  and redraw display.

for c = 1:4
   handles.smooth(c) = 0;      handles.slope(c) = 0;
   handles.pdf(c) = 0;         handles.cdf(c) = 0;
   handles = setfield(handles, ['set' num2str(c)], [0 0; 1 1]);
end
set(handles.smooth_checkbox, 'Value', 0);
set(handles.slope_checkbox, 'Value', 0);
set(handles.pdf_checkbox, 'Value', 0);
set(handles.cdf_checkbox, 'Value', 0);
guidata(hObject, handles);
set(handles.ice, 'Pointer', 'watch');
graph(handles);      render(handles);
set(handles.ice, 'Pointer', 'arrow');

%--------------------------------------------------------------------%
function pdf_checkbox_Callback(hObject, eventdata, handles)
%  Accept PDF (probability density function or histogram) display
%  parameter for currently selected color component and redraw
%  mapping function if smoothing is on. If set, clear CDF display.

if get(hObject, 'Value')
   handles.pdf(handles.cindex) = 1;
   set(handles.cdf_checkbox, 'Value', 0);
   handles.cdf(handles.cindex) = 0;
else
```

```
    handles.pdf(handles.cindex) = 0;
end
guidata(hObject, handles);      graph(handles);

%--------------------------------------------------------------------%
function cdf_checkbox_Callback(hObject, eventdata, handles)
%  Accept CDF (cumulative distribution function) display parameter
%  for selected color component and redraw mapping function if
%  smoothing is on. If set, clear CDF display.

if get(hObject, 'Value')
    handles.cdf(handles.cindex) = 1;
    set(handles.pdf_checkbox, 'Value', 0);
    handles.pdf(handles.cindex) = 0;
else
    handles.cdf(handles.cindex) = 0;
end
guidata(hObject, handles);      graph(handles);

%--------------------------------------------------------------------%
function mapbar_checkbox_Callback(hObject, eventdata, handles)
%  Accept changes to bar map enable state and redraw bars.

handles.barmap = get(hObject, 'Value');
guidata(hObject, handles);      render(handles);

%--------------------------------------------------------------------%
function mapimage_checkbox_Callback(hObject, eventdata, handles)
%  Accept changes to the image map state and redraw image.

handles.imagemap = get(hObject, 'Value');
guidata(hObject, handles);      render(handles);

%--------------------------------------------------------------------%
function component_popup_Callback(hObject, eventdata, handles)
%  Accept color component selection, update component specific
%  parameters on GUI, and draw the selected mapping function.

c = get(hObject, 'Value');
handles.cindex = c;
handles.curve = strcat('set', num2str(c));
guidata(hObject, handles);
set(handles.smooth_checkbox, 'Value', handles.smooth(c));
set(handles.slope_checkbox, 'Value', handles.slope(c));
set(handles.pdf_checkbox, 'Value', handles.pdf(c));
set(handles.cdf_checkbox, 'Value', handles.cdf(c));
graph(handles);

%--------------------------------------------------------------------%
% --- Executes during object creation, after setting all properties.
function component_popup_CreateFcn(hObject, eventdata, handles)
% hObject    handle to component_popup (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until all CreateFcns called
```

```
% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), ...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```



APPENDIX C

Additional Custom M-Functions

Preview

This appendix contains a listing of all the M-functions that are *not* listed earlier in the book. The functions are organized alphabetically. The first two lines of each function are typed in bold letters as a visual cue to facilitate finding the function and reading its summary description. Being part of this book, all the following functions are copyrighted and they are intended to be used exclusively by the individual who owns this copy of the book. Any type of dissemination, including copying in any form and/or posting electronically by any means, such as local servers and the Internet, without written consent from the publisher constitutes a violation of national and international copyright law.

A

```
function f = adpmedian(g, Smax)
%ADPMEDIAN Perform adaptive median filtering.
%   F = ADPMEDIAN(G, SMAX) performs adaptive median filtering of
%   image G.  The median filter starts at size 3-by-3 and iterates
%   up to size SMAX-by-SMAX. SMAX must be an odd integer greater
%   than 1.

% SMAX must be an odd, positive integer greater than 1.
if (Smax <= 1) || (Smax/2 == round(Smax/2)) || (Smax ~= round(Smax))
   error('SMAX must be an odd integer > 1.')
end

% Initial setup.
f = g;
f(:) = 0;
```

```
alreadyProcessed = false(size(g));

% Begin filtering.
for k = 3:2:Smax
   zmin = ordfilt2(g, 1, ones(k, k), 'symmetric');
   zmax = ordfilt2(g, k * k, ones(k, k), 'symmetric');
   zmed = medfilt2(g, [k k], 'symmetric');

   processUsingLevelB = (zmed > zmin) & (zmax > zmed) & ...
       ~alreadyProcessed;
   zB = (g > zmin) & (zmax > g);
   outputZxy  = processUsingLevelB & zB;
   outputZmed = processUsingLevelB & ~zB;
   f(outputZxy) = g(outputZxy);
   f(outputZmed) = zmed(outputZmed);

   alreadyProcessed = alreadyProcessed | processUsingLevelB;
   if all(alreadyProcessed(:))
      break;
   end
end

% Output zmed for any remaining unprocessed pixels. Note that this
% zmed was computed using a window of size Smax-by-Smax, which is
% the final value of k in the loop.
f(~alreadyProcessed) = zmed(~alreadyProcessed);
```

```
function av = average(A)
%AVERAGE Computes the average value of an array.
%   AV = AVERAGE(A) computes the average value of input array, A,
%   which must be a 1-D or 2-D array.

% Check the validity of the input. (Keep in mind that
% a 1-D array is a special case of a 2-D array.)
if ndims(A) > 2
   error('The dimensions of the input cannot exceed 2.')
end

% Compute the average
av = sum(A(:))/length(A(:));
```

B

```
function rc_new = bound2eight(rc)
%BOUND2EIGHT Convert 4-connected boundary to 8-connected boundary.
%   RC_NEW = BOUND2EIGHT(RC) converts a four-connected boundary to an
%   eight-connected boundary.  RC is a P-by-2 matrix, each row of
%   which contains the row and column coordinates of a boundary
%   pixel.  RC must be a closed boundary; in other words, the last
```

```
%   row of RC must equal the first row of RC.  BOUND2EIGHT removes
%   boundary pixels that are necessary for four-connectedness but not
%   necessary for eight-connectedness.  RC_NEW is a Q-by-2 matrix,
%   where Q <= P.

if ~isempty(rc) && ~isequal(rc(1, :), rc(end, :))
   error('Expected input boundary to be closed.');
end

if size(rc, 1) <= 3
   % Degenerate case.
   rc_new = rc;
   return;
end

% Remove last row, which equals the first row.
rc_new = rc(1:end - 1, :);

% Remove the middle pixel in four-connected right-angle turns.  We
% can do this in a vectorized fashion, but we can't do it all at
% once. Similar to the way the 'thin' algorithm works in bwmorph,
% we'll remove first the middle pixels in four-connected turns where
% the row and column are both even; then the middle pixels in the all
% the remaining four-connected turns where the row is even and the
% column is odd; then again where the row is odd and the column is
% even; and finally where both the row and column are odd.

remove_locations = compute_remove_locations(rc_new);
field1 = remove_locations & (rem(rc_new(:, 1), 2) == 0) & ...
         (rem(rc_new(:, 2), 2) == 0);
rc_new(field1, :) = [];

remove_locations = compute_remove_locations(rc_new);
field2 = remove_locations & (rem(rc_new(:, 1), 2) == 0) & ...
         (rem(rc_new(:, 2), 2) == 1);
rc_new(field2, :) = [];

remove_locations = compute_remove_locations(rc_new);
field3 = remove_locations & (rem(rc_new(:, 1), 2) == 1) & ...
         (rem(rc_new(:, 2), 2) == 0);
rc_new(field3, :) = [];

remove_locations = compute_remove_locations(rc_new);
field4 = remove_locations & (rem(rc_new(:, 1), 2) == 1) & ...
         (rem(rc_new(:, 2), 2) == 1);
rc_new(field4, :) = [];

% Make the output boundary closed again.
rc_new = [rc_new; rc_new(1, :)];
```

```
%--------------------------------------------------------------------%
function remove = compute_remove_locations(rc)

% Circular diff.
d = [rc(2:end, :); rc(1, :)] - rc;

% Dot product of each row of d with the subsequent row of d,
% performed in circular fashion.
d1 = [d(2:end, :); d(1, :)];
dotprod = sum(d .* d1, 2);

% Locations of N, S, E, and W transitions followed by
% a right-angle turn.
remove = ~all(d, 2) & (dotprod == 0);

% But we really want to remove the middle pixel of the turn.
remove = [remove(end, :); remove(1:end - 1, :)];

function rc_new = bound2four(rc)
%BOUND2FOUR Convert 8-connected boundary to 4-connected boundary.
%   RC_NEW = BOUND2FOUR(RC) converts an eight-connected boundary to a
%   four-connected boundary.  RC is a P-by-2 matrix, each row of
%   which contains the row and column coordinates of a boundary
%   pixel.  BOUND2FOUR inserts new boundary pixels wherever there is
%   a diagonal connection.

if size(rc, 1) > 1
   % Phase 1: remove diagonal turns, one at a time until they are
   % all gone.
   done = 0;
   rc1 = [rc(end - 1, :); rc];
   while ~done
      d = diff(rc1, 1);
      diagonal_locations = all(d, 2);
      double_diagonals = diagonal_locations(1:end - 1) & ...
          (diff(diagonal_locations, 1) == 0);
      double_diagonal_idx = find(double_diagonals);
      turns = any(d(double_diagonal_idx, :) ~= ...
                  d(double_diagonal_idx + 1, :), 2);
      turns_idx = double_diagonal_idx(turns);
      if isempty(turns_idx)
         done = 1;
      else
         first_turn = turns_idx(1);
         rc1(first_turn + 1, :) = (rc1(first_turn, :) + ...
                                   rc1(first_turn + 2, :)) / 2;
         if first_turn == 1
            rc1(end, :) = rc1(2, :);
         end
      end
```

```
    end
    rc1 = rc1(2:end, :);
end

% Phase 2: insert extra pixels where there are diagonal connections.

rowdiff = diff(rc1(:, 1));
coldiff = diff(rc1(:, 2));

diagonal_locations = rowdiff & coldiff;
num_old_pixels = size(rc1, 1);
num_new_pixels = num_old_pixels + sum(diagonal_locations);
rc_new = zeros(num_new_pixels, 2);

% Insert the original values into the proper locations in the new RC
% matrix.
idx = (1:num_old_pixels)' + [0; cumsum(diagonal_locations)];
rc_new(idx, :) = rc1;

% Compute the new pixels to be inserted.
new_pixel_offsets = [0 1; -1 0; 1 0; 0 -1];
offset_codes = 2 * (1 - (coldiff(diagonal_locations) + 1)/2) + ...
    (2 - (rowdiff(diagonal_locations) + 1)/2);
new_pixels = rc1(diagonal_locations, :) + ...
    new_pixel_offsets(offset_codes, :);

% Where do the new pixels go?
insertion_locations = zeros(num_new_pixels, 1);
insertion_locations(idx) = 1;
insertion_locations = ~insertion_locations;

% Insert the new pixels.
rc_new(insertion_locations, :) = new_pixels;
```

```
function image = bound2im(b, M, N)
%BOUND2IM Converts a boundary to an image.
%   IMAGE = BOUND2IM(b) converts b, an np-by-2 array containing the
%   integer coordinates of a boundary, into a binary image with 1s
%   in the locations of the coordinates in b and 0s elsewhere. The
%   height and width of the image are equal to the Mmin + H and Nmin
%   + W, where Mmin = min(b(:,1)) - 1, N = min(b(:,2)) - 1, and H
%   and W are the height and width of the boundary. In other words,
%   the image created is the smallest image that will encompass the
%   boundary while maintaining the its original coordinate values.
%
%   IMAGE = BOUND2IM(b, M, N) places the boundary in a region of
%   size M-by-N. M and N must satisfy the following conditions:
%
%       M >= max(b(:,1)) - min(b(:,1)) + 1
%       N >= max(b(:,2)) - min(b(:,2)) + 1
```

```
   Typically, M = size(f, 1) and N = size(f, 2), where f is the
   image from which the boundary was extracted. In this way, the
   coordinates of IMAGE and f are registered with respect to each
   other.

Check input.
 size(b, 2) ~= 2
 error('The boundary must be of size np-by-2')
d

Make sure the coordinates are integers.
 = round(b);

Defaults.
 nargin == 1
 Mmin = min(b(:,1)) - 1;
 Nmin = min(b(:,2)) - 1;
 H = max(b(:,1)) - min(b(:,1)) + 1; % Height of boundary.
 W = max(b(:,2)) - min(b(:,2)) + 1; % Width of boundary.
 M = H + Mmin;
 N = W + Nmin;
nd

Create the image.
mage = false(M, N);
inearIndex = sub2ind([M, N], b(:,1), b(:,2));
mage(linearIndex) = 1;

unction [dir, x0 y0] = boundarydir(x, y, orderout)
BOUNDARYDIR Determine the direction of a sequence of planar points.
   [DIR] = BOUNDARYDIR(X, Y) determines the direction of travel of
   a closed, nonintersecting sequence of planar points with
   coordinates contained in column vectors X and Y. Values of DIR
   are 'cw' (clockwise) and 'ccw' (counterclockwise). The direction
   of travel is with respect to the image coordinate system defined
   in Chapter 2 of the book.

   [DIR, XO, YO] = BOUNDARYDIR(X, Y, ORDEROUT) determines the
   direction DIR of the input sequence, and also outputs the
   sequence with its direction of travel as specified in ORDEROUT.
   Valid values of this parameter as 'cw' and 'ccw'. The
   coordinates of the output sequence are column vectors XO and YO.

   The input sequence is assumed to be nonintersecting, and it
   cannot have duplicate points, with the exception of the first
   and last points possibly being the same, a condition often
   resulting from boundary-following functions, such as
   bwboundaries.
```

```
% Preliminaries.
% Make sure coordinates are column vectors.
x = x(:);
y = y(:);

% If the first and last points are the same, delete the last point.
% The point will be restored later.
restore = false;
if x(1) == x(end) && y(1) == y(end)
   x = x(1:end-1);
   y = y(1:end-1);
   restore = true;
end
% Check for duplicate points.
if length([x y]) ~= length(unique([x y],'rows'))
   error('No duplicate points except first and last are allowed.')
end

% The topmost, leftmost point in the sequence is always a convex
% vertex.
x0 = x;
y0 = y;
cx = find(x0 == min(x0));
cy = find(y0 == min(y0(cx)));
x1 = x0(cx(1));
y1 = y0(cy(1));
% Scroll data so that the first point in the sequence is (x1, y1),
% the guaranteed convex point.
I = find(x0 == x1 & y0 == y1);
x0 = circshift(x0, [-(I - 1), 0]);
y0 = circshift(y0, [-(I - 1), 0]);

% Form the matrix needed to check for travel direction. Only three
% points are needed: (x1, y1), the point before it, and the point
% after it.
A = [x0(end) y0(end) 1; x0(1) y0(1) 1; x0(2) y0(2) 1];
dir = 'cw';
if det(A) > 0
   dir = 'ccw';
end

% Prepare outputs.
if nargin == 3
   x0 = x; % Reuse x0 and y0.
   y0 = y;
   if ~strcmp(dir, orderout)
      x0(2:end) = flipud(x0(2:end)); % Reverse order of travel.
      y0(2:end) = flipud(y0(2:end));
   end
   if restore
```

```
      x0(end + 1) = x0(1);
      y0(end + 1) = y0(1);
   end
end

function [s, sUnit] = bsubsamp(b, gridsep)
%BSUBSAMP Subsample a boundary.
%   [S, SUNIT] = BSUBSAMP(B, GRIDSEP) subsamples the boundary B by
%   assigning each of its points to the grid node to which it is
%   closest.  The grid is specified by GRIDSEP, which is the
%   separation in pixels between the grid lines. For example, if
%   GRIDSEP = 2, there are two pixels in between grid lines. So, for
%   instance, the grid points in the first row would be at (1,1),
%   (1,4), (1,6), ..., and similarly in the y direction. The value
%   of GRIDSEP must be an integer. The boundary is specified by a
%   set of coordinates in the form of an np-by-2 array.  It is
%   assumed that the boundary is one pixel thick and that it is
%   ordered in a clockwise or counterclockwise sequence.
%
%   Output S is the subsampled boundary. Output SUNIT is normalized
%   so that the grid separation is unity.  This is useful for
%   obtaining the Freeman chain code of the subsampled boundary. The
%   outputs are in the same order (clockwise or counterclockwise) as
%   the input. There are no duplicate points in the output.

% Check inputs.
[np, nc] = size(b);
if np < nc
   error('b must be of size np-by-2.');
end
if isinteger(gridsep)
   error('gridsep must be an integer.')
end

% Find the maximum span of the boundary.
xmax = max(b(:, 1)) + 1;
ymax = max(b(:, 2)) + 1;

% Determine the integral number of grid lines with gridsep points in
% between them that encompass the intervals [1,xmax], [1,ymax].
GLx = ceil((xmax + gridsep)/(gridsep + 1));
GLy = ceil((ymax + gridsep)/(gridsep + 1));

% Form vector of grid coordinates.
I = 1:GLx;
J = 1:GLy;
% Vector of grid line locations intersecting x-axis.
X(I) = gridsep*I + (I - gridsep);
% Vector of grid line locations intersecting y-axis.
Y(J) = gridsep*J + (J - gridsep);
```

```
[C, R] = meshgrid(Y, X); % See CH 02 regarding function meshgrid.
% Vector of grid all coordinates, arranged as Nunbergridpoints-by-2
% array to match the horizontal dimensions of b. This allows
% computation of distances to be vectorized and thus be much more
% efficient.
V =  [C(1:end); R(1:end)]';

% Compute the distance between every element of b and every element
% of the grid.  See Chapter 13 regarding distance computations.
p = np;
q = size(V, 1);
D = sqrt(sum(abs(repmat(permute(b, [1 3 2]), [1 q 1])...
        - repmat(permute(V, [3 1 2]), [p 1 1])).^2, 3));

% D(i, j) is the distance between the ith row of b and the jth
% row of V. Find the min between each element of b and V.
new_b = zeros(np, 2); % Preallocate memory.
for I = 1:np
   idx = find(D(I,:) == min(D(I,:)), 1); % One min in row I of D.
   new_b(I, :) = V(idx, :);
end

% Eliminate duplicates and keep same order as input.
[s, m] = unique(new_b, 'rows');
s = [s, m];
s = fliplr(s);
s = sortrows(s);
s = fliplr(s);
s = s(:, 1:2);

% Scale to unit grid so that can use directly to obtain Freeman
% chain codes.  The shape does not change.
sUnit = round(s./gridsep) + 1;
```

C

```
function image = changeclass(class, varargin)
%CHANGECLASS changes the storage class of an image.
%  I2 = CHANGECLASS(CLASS, I);
%  RGB2 = CHANGECLASS(CLASS, RGB);
%  BW2 = CHANGECLASS(CLASS, BW);
%  X2 = CHANGECLASS(CLASS, X, 'indexed');

%  Copyright 1993-2002 The MathWorks, Inc.  Used with permission.
%  $Revision: 211 $  $Date: 2006-07-31 14:22:42 -0400 (Mon, 31 Jul
2006) $

switch class
case 'uint8'
   image = im2uint8(varargin{:});
```

```
case 'uint16'
   image = im2uint16(varargin{:});
case 'double'
   image = im2double(varargin{:});
otherwise
   error('Unsupported IPT data class.');
end

function H = cnotch(type, notch, M, N, C, D0, n)
%CNOTCH Generates circularly symmetric notch filters.
%   H = CNOTCH(TYPE, NOTCH, M, N, C, D0, n) generates a notch filter
%   of size M-by-N. C is a K-by-2 matrix with K pairs of frequency
%   domain coordinates (u, v) that define the centers of the filter
%   notches (when specifying filter locations, remember that
%   coordinates in MATLAB run from 1 to M and 1 to N). Coordinates
%   (u, v) are specified for one notch only. The corresponding
%   symmetric notches are generated automatically. D0 is the radius
%   (cut-off frequency) of the notches. It can be specified as a
%   scalar, in which case it is used in all K notch pairs, or it can
%   be a vector of length K, containing an individual cutoff value
%   for each notch pair. n is the order of the Butterworth filter if
%   one is specified.
%
%       Valid values of TYPE are:
%
%           'ideal'     Ideal notchpass filter. n is not used.
%
%           'btw'       Butterworth notchpass filter of order n. The
%                       default value of n is 1.
%
%           'gaussian'  Gaussian notchpass filter. n is not used.
%
%       Valid values of NOTCH are:
%
%           'reject'    Notchreject filter.
%
%           'pass'      Notchpass filter.
%
%       One of these two values must be specified for NOTCH.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).

% Preliminaries.
if nargin < 7
   n = 1; % Default for Butterworth filter.
end

% Define tha largest array of odd dimensions that fits in H. This is
```

```
% required to preserve symmetry in the filter. If necessary, a row
% and/or column is added to the filter at the end of the function.
MO = M;
NO = N;
if iseven(M)
   MO = M - 1;
end
if iseven(N)
   NO = N - 1;
end

% Center of the filter:
center = [floor(MO/2) + 1, floor(NO/2) + 1];

% Number of notch pairs.
K = size(C, 1);
% Cutoff values.
if numel(DO) == 1
        DO(1:K) = DO; % All cut offs are the same.
end

% Shift notch centers  so that they are with respect to the center
% of the filter (and the frequency rectangle).
center = repmat(center, size(C,1), 1);
C = C - center;

% Begin filter computations. All filters are computed as notchreject
% filters. At the end, they are changed to notchpass filters if it
% is so specified in parameter NOTCH.
H = rejectFilter(type, MO, NO, DO, K, C, n);

% Finished. Format the output.
H = processOutput(notch, H, M, N, center);

%-------------------------------------------------------------------%
function H = rejectFilter(type, MO, NO, DO, K, C, n)
% Initialize the filter array to be an "all pass" filter. This
% constant filter is then multiplied by the notchreject filters
% placed at the locations in C with respect to the center of the
% frequency rectangle.
H = ones(MO, NO, 'single');

% Generate filter.
for I = 1:K
   % Place a notch at each location in delta. Function hpfilter
   % returns the filters uncentered. Use fftshit to center the
   % filter at each location. The filters are made larger than
   % M-by-N to simplify indexing in function placeNotches.
   Usize = MO + 2*abs(C(I, 1));
   Vsize = NO + 2*abs(C(I, 2));
```

```
   filt = fftshift(hpfilter(type, Usize , Vsize, DO(I), n));
   % Insert FILT in H.
   H = placeNotches(H, filt, C(I,1), C(I,2));
end

%-----------------------------------------------------------------%
function P = placeNotches(H, filt, delu, delv)
% Places in H the notch contained in FILT.

[M N] = size(H);
U = 2*abs(delu);
V = 2*abs(delv);

% The following calculations are to determine the (common) area of
% overlap between array H and the notch filter FILT.
if delu >= 0 && delv >= 0
   filtCommon = filt(1:M, 1:N); % Displacement is in Q1.
elseif delu < 0 && delv >= 0
   filtCommon = filt(U + 1:U + M, 1:N); % Displacement is in Q2.
elseif delu < 0 && delv < 0
   filtCommon = filt(U + 1:U + M, V + 1:V + N); % Q3
elseif delu >= 0 && delv <= 0
   filtCommon = filt(1:M, V + 1:V + N); % Q4
end

% Compute the product of H and filtCommon. They are registered.
P = ones(M, N).*filtCommon;

% The conjugate notch location is determined by rotating P 180
% degress. This is the same as flipping P left-right and up-down.
% The product of P and its rotated version contain FILT and its
% conjugate.
P = P.*(flipud(fliplr(P)));
P = H.*P; % A new notch and its conjugate were inserted.

%-----------------------------------------------------------------%
function Hout = processOutput(notch, H, M, N, center)
% At this point, H is an odd array in both dimensions (see comments
% at the beginning of the function). In the following, we insert a
% row if M is even, and a column if N is even. The new row and
% column have to be symmetric about their center to preserve
% symmetry in the filter. They are created by duplicating the first
% row and column of H and then making them symmetric.
centerU = center(1,1);
centerV = center(1,2);
newRow = H(1,:);
newRow(1:centerV - 1) = fliplr(newRow(centerV+1:end)); %Symmetric now.
newCol = H(:,1);
newCol(1:centerU - 1) = flipud(newCol(centerU+1:end)); %Symmetric.
% Insert the new row and/or column if appropriate.
```

```
if iseven(M) && iseven(N)
   Hout = cat(1, newRow, H);
   newCol = cat(1, H(1,1), newCol);
   Hout = cat(2, newCol, Hout);
elseif iseven(M) && isodd(N)
   Hout = cat(1, newRow, H);
elseif isodd(M) && iseven(N)
   Hout = cat(2, newCol, H);
else
   Hout = H;
end

% Uncenter the filter, as required for filtering with dftfilt.
Hout = ifftshift(Hout);

% Generate a pass filter if one was specified.
if strcmp(notch, 'pass')
   Hout = 1 - Hout;
end
```

```
function [VG, A, PPG]= colorgrad(f, T)
%COLORGRAD Computes the vector gradient of an RGB image.
%   [VG, VA, PPG] = COLORGRAD(F, T) computes the vector gradient, VG,
%   and corresponding angle array, VA, (in radians) of RGB image
%   F. It also computes PPG, the per-plane composite gradient
%   obtained by summing the 2-D gradients of the individual color
%   planes. Input T is a threshold in the range [0, 1]. If it is
%   included in the argument list, the values of VG and PPG are
%   thresholded by letting VG(x,y) = 0 for values <= T and VG(x,y) =
%   VG(x,y) otherwise. Similar comments apply to PPG.  If T is not
%   included in the argument list then T is set to 0. Both output
%   gradients are scaled to the range [0, 1].

if (ndims(f) ~= 3) || (size(f, 3) ~= 3)
   error('Input image must be RGB.');
end

% Compute the x and y derivatives of the three component images
% using Sobel operators.
sh = fspecial('sobel');
sv = sh';
Rx = imfilter(double(f(:, :, 1)), sh, 'replicate');
Ry = imfilter(double(f(:, :, 1)), sv, 'replicate');
Gx = imfilter(double(f(:, :, 2)), sh, 'replicate');
Gy = imfilter(double(f(:, :, 2)), sv, 'replicate');
Bx = imfilter(double(f(:, :, 3)), sh, 'replicate');
By = imfilter(double(f(:, :, 3)), sv, 'replicate');

% Compute the parameters of the vector gradient.
gxx = Rx.^2 + Gx.^2 + Bx.^2;
```

```
gyy = Ry.^2 + Gy.^2 + By.^2;
gxy = Rx.*Ry + Gx.*Gy + Bx.*By;
A = 0.5*(atan(2*gxy./(gxx - gyy + eps)));
G1 = 0.5*((gxx + gyy) + (gxx - gyy).*cos(2*A) + 2*gxy.*sin(2*A));

% Now repeat for angle + pi/2. Then select the maximum at each point.
A = A + pi/2;
G2 = 0.5*((gxx + gyy) + (gxx - gyy).*cos(2*A) + 2*gxy.*sin(2*A));
G1 = G1.^0.5;
G2 = G2.^0.5;
% Form VG by picking the maximum at each (x,y) and then scale
% to the range [0, 1].
VG = mat2gray(max(G1, G2));

% Compute the per-plane gradients.
RG = sqrt(Rx.^2 + Ry.^2);
GG = sqrt(Gx.^2 + Gy.^2);
BG = sqrt(Bx.^2 + By.^2);
% Form the composite by adding the individual results and
% scale to [0, 1].
PPG = mat2gray(RG + GG + BG);

% Threshold the result.
if nargin == 2
   VG = (VG > T).*VG;
   PPG = (PPG > T).*PPG;
end
```

```
function I = colorseg(varargin)
%COLORSEG Performs segmentation of a color image.
%   S = COLORSEG('EUCLIDEAN', F, T, M) performs segmentation of color
%   image F using a Euclidean measure of similarity. M is a 1-by-3
%   vector representing the average color used for segmentation (this
%   is the center of the sphere in Fig. 6.26 of DIPUM). T is the
%   threshold against which the distances are compared.
%
%   S = COLORSEG('MAHALANOBIS', F, T, M, C) performs segmentation of
%   color image F using the Mahalanobis distance as a measure of
%   similarity. C is the 3-by-3 covariance matrix of the sample color
%   vectors of the class of interest. See function covmatrix for the
%   computation of C and M.
%
%   S is the segmented image (a binary matrix) in which 0s denote the
%   background.

% Preliminaries.
% Recall that varargin is a cell array.
f = varargin{2};
if (ndims(f) ~= 3) || (size(f, 3) ~= 3)
   error('Input image must be RGB.');
end
```

```
M = size(f, 1); N = size(f, 2);
% Convert f to vector format using function imstack2vectors.
f = imstack2vectors(f);
f = double(f);
% Initialize I as a column vector.  It will be reshaped later
% into an image.
I = zeros(M*N, 1);
T = varargin{3};
m = varargin{4};
m = m(:)'; % Make sure that m is a row vector.

if length(varargin) == 4
   method = 'euclidean';
elseif length(varargin) == 5
   method = 'mahalanobis';
else
   error('Wrong number of inputs.');
end

switch method
case 'euclidean'
   % Compute the Euclidean distance between all rows of X and m. See
   % Section 12.2 of DIPUM for an explanation of the following
   % expression. D(i) is the Euclidean distance between vector X(i,:)
   % and vector m.
   p = length(f);
   D = sqrt(sum(abs(f - repmat(m, p, 1)).^2, 2));
case 'mahalanobis'
   C = varargin{5};
   D = mahalanobis(f, C, m);
otherwise
   error('Unknown segmentation method.')
end

% D is a vector of size MN-by-1 containing the distance computations
% from all the color pixels to vector m. Find the distances <= T.
J = find(D <= T);

% Set the values of I(J) to 1.  These are the segmented
% color pixels.
I(J) = 1;

% Reshape I into an M-by-N image.
I = reshape(I, M, N);

function c = connectpoly(x, y)
%CONNECTPOLY Connects vertices of a polygon.
%   C = CONNECTPOLY(X, Y) connects the points with coordinates given
%   in X and Y with straight lines. These points are assumed to be a
%   sequence of polygon vertices organized in the clockwise or
```

```
%   counterclockwise direction. The output, C, is the set of points
%   along the boundary of the polygon in the form of an nr-by-2
%   coordinate sequence in the same direction as the input. The last
%   point in the sequence is equal to the first.

v = [x(:), y(:)];

% Close polygon.
if ~isequal(v(end, :), v(1, :))
   v(end + 1, :) = v(1, :);
end

% Connect vertices.
segments = cell(1, length(v) - 1);
for I = 2:length(v)
   [x, y] = intline(v(I - 1, 1), v(I, 1), v(I - 1, 2), v(I, 2));
   segments{I - 1} = [x, y];
end

c = cat(1, segments{:});

function cp = cornerprocess(c, T, q)
%CORNERPROCESS Processes the output of function cornermetric.
%   CP = CORNERPROCESS(C, T, Q) postprocesses C, the output of
%   function CORNERMETRIC, with the objective of reducing the
%   number of irrelevant corner points (with respect to threshold T)
%   and the number of multiple corners in a neighborhood of size
%   Q-by-Q. If there are multiple corner points contained within
%   that neighborhood, they are eroded morphologically to one corner
%   point.
%
%   A corner point is said to have been found at coordinates (I, J)
%   if C(I,J) > T.
%
%   A good practice is to normalize the values of C to the range [0
%   1], in im2double format before inputting C into this function.
%   This facilitates interpretation of the results and makes
%   thresholding more intuitive.

% Peform thresholding.
cp = c > T;

% Dilate CP to incorporate close neighbors.
B = ones(q);
cp = imdilate(cp, B);

% Shrink connnected components to single points.
cp = bwmorph(cp, 'shrink','Inf');

function cv2tifs(y, f)
%CV2TIFS Decodes a TIFS2CV compressed image sequence.
%   Y = CV2TIFS(Y, F) decodes compressed sequence Y (a structure
```

```
%   generated by TIFS2CV) and creates a multiframe TIFF file F.
%
%   See also TIFS2CV.

% Get the number of frames, block size, and reconstruction quality.
fcnt = double(y.frames);
m = double(y.blksz);
q = double(y.quality);

% Reconstruct the first image in the sequence and store.
if q == 0
    r = double(huff2mat(y.video(1)));
else
    r = double(jpeg2im(y.video(1)));
end
imwrite(uint8(r), f, 'Compression', 'none', 'WriteMode', 'overwrite');

% Get the frame size and motion vectors.
fsz = size(r);
mvsz = [fsz/m 2 fcnt];
mv = int16(huff2mat(y.motion));
mv = reshape(mv, mvsz);

% For frames except the first, get a motion conpensated prediction
% residual and add to the proper reference subimages.
for i = 2:fcnt
    if q == 0
        pe = double(huff2mat(y.video(i)));
    else
        pe = double(jpeg2im(y.video(i)) - 255);
    end
    peC = im2col(pe, [m m], 'distinct');

    for col = 1:size(peC, 2)
        u = 1 + mod(m * (col - 1), fsz(1));
        v = 1 + m * floor((col - 1) * m / fsz(1));
        rx = u - mv(1 + floor((u - 1)/m), 1 + floor((v - 1)/m), ...
            1, i);
        ry = v - mv(1 + floor((u - 1)/m), 1 + floor((v - 1)/m), ...
            2, i);

        subimage = r(rx:rx + m - 1, ry:ry + m - 1);
        peC(:, col) = subimage(:) - peC(:, col);
    end

    r = col2im(double(uint16(peC)), [m m], fsz, 'distinct');
    imwrite(uint8(r), f, 'Compression', 'none', ...
        'WriteMode', 'append');
end
```

D

```
function s = diameter(L)
%DIAMETER Measure diameter and related properties of image regions.
%   S = DIAMETER(L) computes the diameter, the major axis endpoints,
%   the minor axis endpoints, and the basic rectangle of each labeled
%   region in the label matrix L. Positive integer elements of L
%   correspond to different regions. For example, the set of elements
%   of L equal to 1 corresponds to region 1; the set of elements of L
%   equal to 2 corresponds to region 2; and so on.  S is a structure
%   array of length max(L(:)). The fields of the structure array
%   include:
%
%     Diameter
%     MajorAxis
%     MinorAxis
%     BasicRectangle
%
%   The Diameter field, a scalar, is the maximum distance between any
%   two pixels in the corresponding region.
%
%   The MajorAxis field is a 2-by-2 matrix.  The rows contain the row
%   and column coordinates for the endpoints of the major axis of the
%   corresponding region.
%
%   The MinorAxis field is a 2-by-2 matrix.  The rows contain the row
%   and column coordinates for the endpoints of the minor axis of the
%   corresponding region.
%
%   The BasicRectangle field is a 4-by-2 matrix.  Each row contains
%   the row and column coordinates of a corner of the
%   region-enclosing rectangle defined by the major and minor axes.
%
%   For more information about these measurements, see Section 11.2.1
%   of Digital Image Processing, by Gonzalez and Woods, 2nd edition,
%   Prentice Hall.

s = regionprops(L, {'Image', 'BoundingBox'});

for k = 1:length(s)
    [s(k).Diameter, s(k).MajorAxis, perim_r, perim_c] = ...
        compute_diameter(s(k));
    [s(k).BasicRectangle, s(k).MinorAxis] = ...
        compute_basic_rectangle(s(k), perim_r, perim_c);
end

%--------------------------------------------------------------------%
function [d, majoraxis, r, c] = compute_diameter(s)
%   [D, MAJORAXIS, R, C] = COMPUTE_DIAMETER(S) computes the diameter
%   and major axis for the region represented by the structure S. S
```

```
%   must contain the fields Image and BoundingBox.  COMPUTE_DIAMETER
%   also returns the row and column coordinates (R and C) of the
%   perimeter pixels of s.Image.

% Compute row and column coordinates of perimeter pixels.
[r, c] = find(bwperim(s.Image));
r = r(:);
c = c(:);
[rp, cp] = prune_pixel_list(r, c);

num_pixels = length(rp);
switch num_pixels
case 0
   d = -Inf;
   majoraxis = ones(2, 2);

case 1
   d = 0;
   majoraxis = [rp cp; rp cp];

case 2
   d = (rp(2) - rp(1))^2 + (cp(2) - cp(1))^2;
   majoraxis = [rp cp];

otherwise
   % Generate all combinations of 1:num_pixels taken two at at time.
   % Method suggested by Peter Acklam.
   [idx(:, 2) idx(:, 1)] = find(tril(ones(num_pixels), -1));
   rr = rp(idx);
   cc = cp(idx);

   dist_squared = (rr(:, 1) - rr(:, 2)).^2 + ...
       (cc(:, 1) - cc(:, 2)).^2;
   [max_dist_squared, idx] = max(dist_squared);
   majoraxis = [rr(idx,:)' cc(idx,:)'];

   d = sqrt(max_dist_squared);

   upper_image_row = s.BoundingBox(2) + 0.5;
   left_image_col = s.BoundingBox(1) + 0.5;

   majoraxis(:, 1) = majoraxis(:, 1) + upper_image_row - 1;
   majoraxis(:, 2) = majoraxis(:, 2) + left_image_col - 1;
end

%--------------------------------------------------------------------%
function [basicrect, minoraxis] = compute_basic_rectangle(s, ...
                                                    perim_r, perim_c)
%   [BASICRECT,MINORAXIS] = COMPUTE_BASIC_RECTANGLE(S, PERIM_R,
%   PERIM_C) computes the basic rectangle and the minor axis
```

```
%   end-points for the region represented by the structure S.  S must
%   contain the fields Image, BoundingBox, MajorAxis, and
%   Diameter. PERIM_R and PERIM_C are the row and column coordinates
%   of perimeter of s.Image. BASICRECT is a 4-by-2 matrix, each row
%   of which contains the row and column coordinates of one corner of
%   the basic rectangle.

% Compute the orientation of the major axis.
theta = atan2(s.MajorAxis(2, 1) - s.MajorAxis(1, 1), ...
              s.MajorAxis(2, 2) - s.MajorAxis(1, 2));

% Form rotation matrix.
T = [cos(theta) sin(theta); -sin(theta) cos(theta)];

% Rotate perimeter pixels.
p = [perim_c perim_r];
p = p * T';

% Calculate minimum and maximum x- and y-coordinates for the rotated
% perimeter pixels.
x = p(:, 1);
y = p(:, 2);
min_x = min(x);
max_x = max(x);
min_y = min(y);
max_y = max(y);

corners_x = [min_x max_x max_x min_x]';
corners_y = [min_y min_y max_y max_y]';

% Rotate corners of the basic rectangle.
corners = [corners_x corners_y] * T;

% Translate according to the region's bounding box.
upper_image_row = s.BoundingBox(2) + 0.5;
left_image_col = s.BoundingBox(1) + 0.5;

basicrect = [corners(:, 2) + upper_image_row - 1, ...
             corners(:, 1) + left_image_col - 1];

% Compute minor axis end-points, rotated.
x = (min_x + max_x) / 2;
y1 = min_y;
y2 = max_y;
endpoints = [x y1; x y2];

% Rotate minor axis end-points back.
endpoints = endpoints * T;

% Translate according to the region's bounding box.
minoraxis = [endpoints(:, 2) + upper_image_row - 1, ...
```

```
                endpoints(:, 1) + left_image_col - 1];

%--------------------------------------------------------------------%
function [r, c] = prune_pixel_list(r, c)
%   [R, C] = PRUNE_PIXEL_LIST(R, C) removes pixels from the vectors
%   R and C that cannot be endpoints of the major axis.  This
%   elimination is based on geometrical constraints described in
%   Russ, Image Processing Handbook, Chapter 8.

top = min(r);
bottom = max(r);
left = min(c);
right = max(c);

% Which points are inside the upper circle?
x = (left + right)/2;
y = top;
radius = bottom - top;
inside_upper = ( (c - x).^2 + (r - y).^2 ) < radius^2;

% Which points are inside the lower circle?
y = bottom;
inside_lower = ( (c - x).^2 + (r - y).^2 ) < radius^2;

% Which points are inside the left circle?
x = left;
y = (top + bottom)/2;
radius = right - left;
inside_left = ( (c - x).^2 + (r - y).^2 ) < radius^2;

% Which points are inside the right circle?
x = right;
inside_right = ( (c - x).^2 + (r - y).^2 ) < radius^2;

% Eliminate points that are inside all four circles.
delete_idx = find(inside_left & inside_right & ...
                  inside_upper & inside_lower);
r(delete_idx) = [];
c(delete_idx) = [];
```

F

```
function  c = fchcode(b, conn, dir)
%FCHCODE Computes the Freeman chain code of a boundary.
%   C = FCHCODE(B) computes the 8-connected Freeman chain code of a
%   set of 2-D coordinate pairs contained in B, an np-by-2 array. C
%   is a structure with the following fields:
%
%     c.fcc    = Freeman chain code (1-by-np)
```

```
%     c.diff   = First difference of code c.fcc (1-by-np)
%     c.mm     = Integer of minimum magnitude from c.fcc (1-by-np)
%     c.diffmm = First difference of code c.mm (1-by-np)
%     c.x0y0   = Coordinates where the code starts (1-by-2)
%
%   C = FCHCODE(B, CONN) produces the same outputs as above, but
%   with the code connectivity specified in CONN. CONN can be 8 for
%   an 8-connected chain code, or CONN can be 4 for a 4-connected
%   chain code. Specifying CONN = 4 is valid only if the input
%   sequence, B, contains transitions with values 0, 2, 4, and 6,
%   exclusively. If it does not, an error is issued. See table
%   below.
%
%   C = FHCODE(B, CONN, DIR) produces the same outputs as above,
%   but, in addition, the desired code direction is specified.
%   Values for DIR can be:
%
%     'same'      Same as the order of the sequence of points in b.
%                 This is the default.
%
%     'reverse'   Outputs the code in the direction opposite to the
%                 direction of the points in B.  The starting point
%                 for each DIR is the same.
%
%   The elements of B are assumed to correspond to a 1-pixel-thick,
%   fully-connected, closed boundary. B cannot contain duplicate
%   coordinate pairs, except in the first and last positions, which
%   is a common feature of boundary tracing programs.
%
%   FREEMAN CHAIN CODE REPRESENTATION The table on the left shows
%   the 8-connected Freeman chain codes corresponding to allowed
%   deltax, deltay pairs. An 8-chain is converted to a 4-chain if
%   (1) conn = 4; and (2) only transitions 0, 2, 4, and 6 occur in
%   the 8-code.  Note that dividing 0, 2, 4, and 6 by 2 produce the
%   4-code. See Fig. 12.2 for an explanation of the directional 4-
%   and 8-codes.
%
%       ----------------------  ----------------
%       deltax | deltay | 8-code  corresp 4-code
%       ----------------------  ----------------
%         0        1       0           0
%        -1        1       1
%        -1        0       2           1
%        -1       -1       3
%         0       -1       4           2
%         1       -1       5
%         1        0       6           3
%         1        1       7
%       ----------------------  ----------------
%
```

```
%   The formula z = 4*(deltax + 2) + (deltay + 2) gives the
%   following sequence corresponding to rows 1-8 in the preceding
%   table: z = 11,7,6,5,9,13,14,15. These values can be used as
%   indices into the table, improving the speed of computing the
%   chain code. The preceding formula is not unique, but it is based
%   on the smallest integers (4 and 2) that are powers of 2.

% Preliminaries.
if nargin == 1
   dir = 'same';
   conn = 8;
elseif nargin == 2
   dir = 'same';
elseif nargin == 3
   % Nothing to do here.
else
   error('Incorrect number of inputs.')
end
[np, nc] = size(b);
if np < nc
   error('B must be of size np-by-2.');
end

% Some boundary tracing programs, such as bwboundaries.m, output a
% sequence in which the coordinates of the first and last points are
% the same. If this is the case, eliminate the last point.
if isequal(b(1, :), b(np, :))
   np = np - 1;
   b = b(1:np, :);
end

% Build the code table using the single indices from the formula
% for z given above:
C(11)=0; C(7)=1; C(6)=2; C(5)=3; C(9)=4;
C(13)=5; C(14)=6; C(15)=7;

% End of Preliminaries.

% Begin processing.
x0 = b(1, 1);
y0 = b(1, 2);
c.x0y0 = [x0, y0];

% Check the curve for out-of-order points or breaks.
% Get the deltax and deltay between successive points in b. The
% last row of a is the first row of b.
a = circshift(b, [-1, 0]);

% DEL = a - b is an nr-by-2 matrix in which the rows contain the
% deltax and deltay between successive points in b. The two
% components in the kth row of matrix DEL are deltax and deltay
```

```
% between point (xk, yk) and (xk+1, yk+1).  The last row of DEL
% contains the deltax and deltay between (xnr, ynr) and (x1, y1),
% (i.e., between the last and first points in b).
DEL = a - b;

% If the abs value of either (or both) components of a pair
% (deltax, deltay) is greater than 1, then by definition the curve
% is broken (or the points are out of order), and the program
% terminates.
if any(abs(DEL(:, 1)) > 1) || any(abs(DEL(:, 2)) > 1);
   error('The input curve is broken or points are out of order.')
end

% Create a single index vector using the formula described above.
z = 4*(DEL(:, 1) + 2) + (DEL(:, 2) + 2);

% Use the index to map into the table. The following are
% the Freeman 8-chain codes, organized in a 1-by-np array.
fcc = C(z);

% Check if direction of code sequence needs to be reversed.
if strcmp(dir, 'reverse')
   fcc = coderev(fcc); % See below for function coderev.
end

% If 4-connectivity is specified, check that all components
% of fcc are 0, 2, 4, or 6.
if conn == 4
   if isempty(find(fcc == 1 || fcc == 3 || fcc == 5 ...
                   || fcc ==7 , 1))
       fcc = fcc./2;
    else
       error('The specified 4-connected code cannot be satisfied.')
   end
end

% Freeman chain code for structure output.
c.fcc = fcc;

% Obtain the first difference of fcc.
c.diff = codediff(fcc,conn); % See below for function codediff.

% Obtain code of the integer of minimum magnitude.
c.mm = minmag(fcc); % See below for function minmag.

% Obtain the first difference of fcc
c.diffmm = codediff(c.mm, conn);

%-------------------------------------------------------------------%
function cr = coderev(fcc)
%   Traverses the sequence of 8-connected Freeman chain code fcc in
```

```
%   the opposite direction, changing the values of each code
%   segment. The starting point is not changed. fcc is a 1-by-np
%   array.

% Flip the array left to right.  This redefines the starting point
% as the last point and reverses the order of "travel" through the
% code.
cr = fliplr(fcc);

% Next, obtain the new code values by traversing the code in the
% opposite direction. (0 becomes 4, 1 becomes 5, ... , 5 becomes 1,
% 6 becomes 2, and 7 becomes 3).
ind1 = find(0 <= cr & cr <= 3);
ind2 = find(4 <= cr & cr <= 7);
cr(ind1) = cr(ind1) + 4;
cr(ind2) = cr(ind2) - 4;

%-------------------------------------------------------------------%
function z = minmag(c)
%      Finds the integer of minimum magnitude in a given
%   4- or 8-connected Freeman chain code, C. The code is assumed to
%   be a 1-by-np array.

% The integer of minimum magnitude starts with min(c), but there
% may be more than one such value. Find them all,
I = find(c == min(c));
% and shift each one left so that it starts with min(c).
J = 0;
A = zeros(length(I), length(c));
for k = I;
   J = J + 1;
   A(J, :) = circshift(c,[0 -(k - 1)]);
end

% Matrix A contains all the possible candidates for the integer of
% minimum magnitude. Starting with the 2nd column, succesively find
% the minima in each column of A. The number of candidates decreases
% as the seach moves to the right on A.  This is reflected in the
% elements of J.  When length(J) = 1, one candidate remains.  This
% is the integer of minimum magnitude.
[M, N] = size(A);
J = (1:M)';
D(J, 1) = 0; % Reserve memory space for loop.
for k = 2:N
   D(1:M, 1) = Inf;
   D(J, 1) = A(J, k);
   amin = min(A(J, k));
   J = find(D(:, 1) == amin);
   if length(J)==1
      z = A(J, :);
      return
```

```
   end
end

%--------------------------------------------------------------------%
function d = codediff(fcc, conn)
%   Computes the first difference of code, FCC. The code FCC is
%   treated as a circular sequence, so the last element of D is the
%   difference between the last and first elements of FCC.  The
%   input code is a 1-by-np vector.

% The first difference is found by counting the number of direction
% changes (in a counter-clockwise direction) that separate two
% adjacent elements of the code.
sr = circshift(fcc, [0, -1]); % Shift input left by 1 location.
delta = sr - fcc;
d = delta;
I = find(delta < 0);

type = conn;
switch type
case 4 % Code is 4-connected
   d(I) = d(I) + 4;
case 8 % Code is 8-connected
   d(I) = d(I) + 8;
end
```

G

```
function v = gmean(A)
%GMEAN Geometric mean of columns.
%   V = GMEAN(A) computes the geometric mean of the columns of A.  V
%   is a row vector with size(A,2) elements.
%
%   Sample M-file used in Chapter 3.

m = size(A, 1);
v = prod(A, 1) .^ (1/m);

function g = gscale(f, varargin)
%GSCALE Scales the intensity of the input image.
%   G = GSCALE(F, 'full8') scales the intensities of F to the full
%   8-bit intensity range [0, 255].  This is the default if there is
%   only one input argument.
%
%   G = GSCALE(F, 'full16') scales the intensities of F to the full
%   16-bit intensity range [0, 65535].
%
%   G = GSCALE(F, 'minmax', LOW, HIGH) scales the intensities of F to
%   the range [LOW, HIGH]. These values must be provided, and they
%   must be in the range [0, 1], independently of the class of the
```

```
%   input. GSCALE performs any necessary scaling. If the input is of
%   class double, and its values are not in the range [0, 1], then
%   GSCALE scales it to this range before processing.
%
%   The class of the output is the same as the class of the input.

if length(varargin) == 0 % If only one argument it must be f.
   method = 'full8';
else
   method = varargin{1};
end

if strcmp(class(f), 'double') & (max(f(:)) > 1 || min(f(:)) < 0)
   f = mat2gray(f);
end

% Perform the specified scaling.
switch method
case 'full8'
   g = im2uint8(mat2gray(double(f)));
case 'full16'
   g = im2uint16(mat2gray(double(f)));
case 'minmax'
   low = varargin{2}; high = varargin{3};
   if low > 1 || low < 0 || high > 1 || high < 0
      error('Parameters low and high must be in the range [0, 1].')
   end
   if strcmp(class(f), 'double')
      low_in = min(f(:));
      high_in = max(f(:));
   elseif strcmp(class(f), 'uint8')
      low_in = double(min(f(:)))./255;
      high_in = double(max(f(:)))./255;
   elseif strcmp(class(f), 'uint16')
      low_in = double(min(f(:)))./65535;
      high_in = double(max(f(:)))./65535;
   end
   % imadjust automatically matches the class of the input.
   g = imadjust(f, [low_in high_in], [low high]);
otherwise
   error('Unknown method.')
end
```

I

```
function P = i2percentile(h, I)
%I2PERCENTILE Computes a percentile given an intensity value.
%   P = I2PERCENTILE(H, I) Given an intensity value, I, and a
%   histogram, H, this function computes the percentile, P, that I
%   represents for the population of intensities governed by
```

```
%   histogram H. I must be in the range [0, 1], independently of the
%   class of the image from which the histogram was obtained. P is
%   returned as a value in the range [0 1]. To convert it to a
%   percentile multiply it by 100. By definition, I = 0 represents
%   the 0th percentile and I = 1 represents 100th percentile.
%
%   Example:
%
%   Suppose that h is a uniform histogram of an uint8 image. Typing
%
%       P = i2percentile(h, 127/255)
%
%   would return P = 0.5, indicating that the input intensity
%   is in the 50th percentile.
%
%   See also function percentile2i.

% Normalized the histogram to unit area. If it is already normalized
% the following computation has no effect.
h = h/sum(h);

% Calculations.
K = numel(h) - 1;
C = cumsum(h); % Cumulative distribution.
if I < 0 || I > 1
    error('Input intensity must be in the range [0, 1].')
elseif I == 0
   P = 0; % Per the definition of percentile.
elseif I == 1
   P = 1; % Per the definition of percentile.
else
   idx = floor(I*K) + 1;
   P = C(idx);
end
```

```
function [X, Y, R] = im2minperpoly(B, cellsize)
%IM2MINPERPOLY Minimum perimeter polygon.
%   [X, Y, R] = IM2MINPERPOLY(B, CELLSIZE) outputs in column vectors
%   X and Y the coordinates of the vertices of the minimum perimeter
%   polygon circumscribing a single binary region or a
%   (nonintersecting) boundary contained in image B. The background
%   in B must be 0, and the region or boundary must have values
%   equal to 1. If instead of an image, B, a list of ordered
%   vertices is available, link the vertices using function
%   connectpoly and then use function bound2im to generate a binary
%   image B containing the boundary.
%
%   R is the region extracted from the image, from which the MPP
%   will be computed (see Figs. 12.5(c) and 12.6(e)). Displaying
%   this region is a good approach to determine interactively a
```

```
%   satisfactory value for CELLSIZE. Parameter CELLSIZE is the size
%   of the square cells that enclose the boundary of the region in
%   B. The value of CELLSIZE must be a positive integer greater than
%   1. See Section 12.2.2 in the book for further details on this
%   parameter, as well as a description and references for the
%   algorithm.

% Preliminaries.
if cellsize <= 1
    error('cellsize must be an integer > 1.');
end
% Check to see that there is only one object in B.
[B, num] = bwlabel(B);
if num > 1
   error('Input image cannot contain more than one region.')
end

% Extract the 4-connected region encompassed by the cellular
% complex. See Fig. 12.6(e) in DIPUM 2/e.
R = cellcomplex(B, cellsize);

% Find the vertices of the MPP.
[X Y] = mppvertices(R, cellsize);

%--------------------------------------------------------------------%
function R = cellcomplex(B, cellsize)
% Computes the cellular complex surrounding a single object in
% binary image B, and outputs in R the region bpounded by the
% cellular complex, as explained in DIPUM/2E Figs. 12.5(c) and
% 12.6(e). Parameter CELLSIZE is as explained earlier.

% Fill the image in case it has holes and compute the 4-connected
% boundary of the result. This guarantees that will be working with
% a single 4-connected boundary, as required by the MPP algorithm.
% Recall that in function bwperim connectivity is with respect to
% the background; therefore, we specify a connectivity of 8 to get a
% connectivity of 4 in the boundary.
B = imfill(B, 'holes');
B = bwperim(B, 8);
[M, N] = size(B);

% Increase image size so that the image is of size K-by-K
% with (a) K >= max(M,N), and (b)  K/cellsize = a power of 2.
K = nextpow2(max(M, N)/cellsize);
K = (2^K)*cellsize;

% Increase image size to the nearest integer power of 2, by
% appending zeros to the end of the image. This will allow
% quadtree  decompositions as small as cells of size 2-by-2,
% which is the smallest allowed value of cellsize.
M1 = K - M;
```

```
N1 = K - N;
B = padarray(B, [M1 N1], 'post'); % B is now of size K-by-K

% Quadtree decomposition.
Q = qtdecomp(B, 0, cellsize);

% Get all the subimages of size cellsize-by-cellsize.
[vals, r, c] = qtgetblk(B, Q, cellsize);

% Find all the subimages that contain at least one black pixel.
% These will be the cells of the cellular complex enclosing the
% boundary.
I = find(sum(sum(vals(:, :, :)) >= 1));
LI = length(I);
x = r(I);
y = c(I);

% [x', y'] is an LI-by-2 array.  Each member of this array  is the
% left, top corner of a black cell of size cellsize-by-cellsize.
% Fill the cells with black to form a closed border of black cells
% around interior points. These are the cells are the cellular
% complex.
for k = 1:LI
   B(x(k):x(k) + cellsize - 1, y(k):y(k) + cellsize - 1) = 1;
end
BF = imfill(B, 'holes');

% Extract the points interior to the cell border. This is the
% region, R, around which the MPP will be found.
B = BF & (~B);
R = B(1:M, 1:N); % Remove the padding and output the region.

%--------------------------------------------------------------------%
function [X, Y] = mppvertices(R, cellsize)
%   Outputs in column vectors X and Y the coordinates of the
%   vertices of the minimum-perimeter polygon that circumscribes
%   region R. This is the region bounded by the cellular complex. It
%   is assumed that the coordinate system used is as defined in
%   Chapter 2 of the book, in which the origin is at the top, left,
%   the positive x-axis extends vertically down from the origin and
%   the positive y-axis extends horizontally to the right. No
%   duplicate vertices are allowed. Parameter CELLSIZE is as
%   explained earlier.

% Extract the 4-connected boundary of the region. Reuse variable B.
% It will be a boundary now. See Fig. 12.6(f) in DIPUM 2/e.
B = bwboundaries(R, 4, 'noholes');
B = B{1};
% Function bwboundaries outputs the last coordinate pair equal
% to the first.  Delete it.
```

```
B = B(1:end - 1, :);

% Obtain the xy coordinates of the boundary. These are column
% vectors.
x = B(:, 1);
y = B(:, 2);

% Format the vertices in the form required by the algorithm.
L = vertexlist(x, y, cellsize);
NV = size(L, 1); % Number of vertices in L.
count = 1;       % Index for the vertices in the list.
k = 1;           % Index for vertices in the MPP.
X(1) = L(1,1);   % 1st vertex, known to be an MPP vertex.
Y(1) = L(1,2);

% Find the vertices of the MPP.
% Initialize.
cMPPV = [L(1,1), L(1,2)]; % Current MPP vertex.
cV = cMPPV;               % Current vertex.
classV = L(1,3);          % Class of current vertex (+1 for convex).
cWH = cMPPV;              % Current WHITE crawler.
cBL = cMPPV;              % Current BLACK crawler.

% Process the vertices. This is the core of the MPP algorithm.
% Note: Cannot preallocate memory for X and Y because their length
% is variable.
while true
   count = count + 1;
   if count > NV + 1
      break;
   end
   % Process next vertex.
   if count == NV + 1  % Have arrived at first vertex again.
      cV = [L(1,1), L(1,2)];
      classV = L(1,3);
   else
      cV = [L(count, 1), L(count, 2)];
      classV = L(count, 3);
   end
   [I, newMPPV, W, B] = mppVtest(cMPPV, cV, classV, cWH, cBL);
   if I == 1 % New MPP vertex found;
      cMPPV = newMPPV;
      K = find(L(:,1) == newMPPV(:, 1) & L(:,2) == newMPPV(:, 2));
      count = K; % Restart at current location of MPP vertex.
      cWH = newMPPV;
      cBL = newMPPV;
      k = k + 1;
      % Vertices of the MPP just found.
      X(k) = newMPPV(1,1);
      Y(k) = newMPPV(1,2);
```

```
    else
        cWH = W;
        cBL = B;
    end
end
% Convert to columns.
X = X(:);
Y = Y(:);

%-----------------------------------------------------------------%
function L = vertexlist(x, y, cellsize)
%   Given a set of coordinates contained in vectors X and Y, this
%   function outputs a list, L, of the form L = [X(k) Y(k) C(k)]
%   where C(k) determines whether X(k) and Y(k) are the coordinates
%   of the apex of a convex, concave, or 180-degree angle. That is,
%   C(k) = 1 if the coordinates (x(k - 1) y(k - 1), (x(k), y(k)) and
%   (x(k + 1), y(k + 1)) form a convex angle; C(k) = -1 if the angle
%   is concave; and C(k) = 0 if the three points are collinear.
%   Concave angles are replaced by their corresponding convex angles
%   in the outer wall for later use in the minimum-perimeter polygon
%   algorithm, as explained in the book.

% Preprocess the input data. First, arrange the the points so that
% the first point is the top, left-most point in the sequence. This
% guarantees that the first vertex of the polygon is convex.
cx = find(x == min(x));
cy = find(y == min(y(cx)));
x1 = x(cx(1));
y1 = y(cy(1));
% Scroll data so that the first point in the sequence is (x1, y1)
I = find(x == x1 & y == y1);
x = circshift(x, [-(I - 1), 0]);
y = circshift(y, [-(I - 1), 0]);

% Next keep only the points at which a change in direction takes
% place. These are the only points that are polygon vertices. Note
% that we cannot preallocate memory for the loop because xnew and
% ynew are of variable length.
J = 1;
K = length(x);
xnew(1) = x(1);
ynew(1) = y(1);
x(K + 1) = x(1);
y(K + 1) = y(1);
for k = 2:K
    s = vsign([x(k - 1),y(k - 1)], [x(k),y(k)], [x(k + 1),y(k + 1)]);
    if s ~= 0
        J = J + 1;
        xnew(J) = x(k); %#ok<AGROW>
        ynew(J) = y(k); %#ok<AGROW>
```

```
    end
end
% Reuse x and y.
x = xnew;
y = ynew;

% The mpp algorithm works with boundaries in the ccw direction.
% Force the sequence to be in that direction. Output dir is the
% direction of the original boundary. It is not used in this
% function.
[dir, x, y] = boundarydir(x, y, 'ccw');

% Obtain the list of vertices.
% Initialize.
K = length(x);
L(:, :, :) = [x(:) y(:) zeros(K,1)]; % Initialize the list.
C = zeros(K, 1); % Preallocate memory for use in a loop later.

% Do the first and last vertices separately.
% First vertex.
s = vsign([x(K) y(K)], [x(1) y(1)], [x(2) y(2)]);
if s > 0
    C(1) = 1;
elseif s < 0
    C(1) = -1;
    [rx ry] = vreplacement([x(K) y(K)], [x(1) y(1)],...
                           [x(2) y(2)], cellsize);
    L(1, 1) = rx;
    L(1, 2) = ry;
else
    C(1) = 0;
end
% Last vertex.
s = vsign([x(K - 1) y(K - 1)], [x(K) y(K)], [x(1) y(1)]);
if s > 0
    C(K) = 1;
elseif s < 0
    C(K) = -1;
    [rx ry] = vreplacement([x(K - 1) y(K - 1)], [x(K) y(K)], ...
                           [x(1) y(1)], cellsize);
    L(K, 1) = rx;
    L(K, 2) = ry;
else
    C(K) = 0;
end

% Process the rest of the vertices.
for k = 2:K - 1
    s = vsign([x(k - 1) y(k - 1)], [x(k) y(k)], [x(k + 1) y(k + 1)]);
     if s > 0
```

```
      C(k) = 1;
    elseif s < 0
      C(k) = -1;
       [rx ry] = vreplacement([x(k - 1) y(k - 1)], [x(k) y(k)], ...
                        [x(k + 1) y(k + 1)], cellsize);
       L(k, 1) = rx;
       L(k, 2) = ry;
    else
      C(k) = 0;
    end
end

% Update the list with the C's.
L(:, 3)= C(:);

%-------------------------------------------------------------------%
function s = vsign(v1, v2, v3)
%   This function etermines whether a vertex V3 is on the
%   positive or the negative side of straight line passing through
%   V1 and V2, or whether the three points are colinear.  V1, V2,
%   and V3 are 1-by-2 or 2-by-1 vectors containing the [x  y]
%   coordinates of the vertices.  If V3 is on the positive side of
%   the line passing through V1 and V2, then the sign is positive (S
%   > 0), if it is on the negative side of the line the sign is
%   negative (S < 0).  If the points are collinear, then S = 0.
%   Another important interpretation is that if the triplet (V1, V2,
%   V3) form a counterclockwise sequence, then S > 0; if the points
%   form a clockwise sequence then S < 0; if the points are
%   collinear, then S = 0.
%
%   The coordinate system is assumed to be the system is as defined
%   in Chapter 2 of the book.
%
%   This function is based in the result from matrix theory that if
%   we arrange the coordinates of the vertices as the matrix
%
%        A = [V1(1) V1(2) 1; V2(1) V2(2) 1; V3(1) V3(2) 1]
%
%   then, S = det(A) has the properties described above, assuming
%   the stated coordinate system and direction of travel.

% Form the matrix on which the test if based:
A = [v1(1) v1(2) 1; v2(1) v2(2) 1; v3(1), v3(2), 1];
% Compute the determinant.
s = det(A);

%-------------------------------------------------------------------%
function [rx ry] = vreplacement(v1, v, v2, cellsize)
%   This function replaces the coordinates V(1) and V(2) of concave
%   vertex V by its diagonal mirror coordinates [RX, RY]. The values
```

```
%   RX and RY depend on the orientation of the triplet (V1, V, V2).
%   V1 is the vertex preceding V and V2 is the vertex following it.
%   All Vs are 1-by-2 or 2-by-1 arrays containing the coordinates of
%   the vertices. It is assumed that the triplet (V1, V, V2) was
%   generated by traveling in the counterclockwise direction, in the
%   coordinate system defined in Chapter 2 of the book, in which the
%   origin is at the top left, the positive x-axis extends down and
%   the positive y-axis extends to the right. Parameter CELLSIZE is
%   as explained earlier.

% Perform the replacement.

if v(1)>v1(1) && v(2) == v1(2) && v(1) == v2(1) && v(2)>v2(2)
    rx = v(1) - cellsize;
    ry = v(2) - cellsize;
elseif v(1) == v1(1) && v(2) > v1(2) && v(1) < v2(1) && ...
    v(2) == v2(2)
    rx = v(1) + cellsize;
    ry = v(2) - cellsize;
elseif v(1) < v1(1) && v(2) == v1(2) && v(1) == v2(1) &&...
    v(2) < v2(2)
    rx = v(1) + cellsize;
    ry = v(2) + cellsize;
elseif v(1) == v1(1) && v(2) < v1(2) && v(1) > v2(1) &&...
    v(2)== v2(2)
    rx = v(1) - cellsize;
    ry = v(2) + cellsize;
else
   % Only the preceding forms are valid arrangements of vertices.
   error('Vertex configuration is not valid.')
end

%-------------------------------------------------------------------%
function [I, newMPPV, W, B] = mppVtest(cMPPV, cV, classcV, cWH, cBL)
%       This function performs tests for existence of an MPP vertex.
%       The parameters are as follows (all except I and class_c_V) are
%       coordinate pairs of the form [x y]).
%     cMPPV          Current MPP vertex (the last MPP vertex found).
%     cV             Current vertex in the sequence.
%     classcV        Class of current vertex (+1 for convex
%                    and -1 for concave).
%     cWH            The current WHITE (convex) vertex.
%     cBL            The current BLACK (concave) vertex
%     I              If I = 1, a new MPP vertex was found
%     newMPPV        Next MPP vertex (if I = 1).
%     W              Next coordinates of WHITE.
%     B              Next coordinates of BLACK.
%
%   The details of the test are explained in Chapter 12 of the book.
```

```
% Preliminaries
I = 0;
newMPPV = [0 0];
W = cWH;
B = cBL;
sW = vsign(cMPPV, cWH, cV);
sB = vsign(cMPPV, cBL, cV);

% Perform test.
if sW > 0
   I = 1; % New MPP vertex found.
   newMPPV = cWH;
   W = newMPPV;
   B = newMPPV;
elseif sB < 0
   I = 1; % New MPP vertex found.
   newMPPV = cBL;
   W = newMPPV;
   B = newMPPV;
elseif (sW <= 0) && (sB >= 0)
   if classcV == 1
      W = cV;
   else
      B = cV;
   end
end
```

```
function [p, pmax, pmin, pn] = improd(f, g)
%IMPROD Compute the product of two images.
%   [P, PMAX, PMIN, PN] = IMPROD(F, G) outputs the element-by-element
%   product of two input images, F and G, the product maximum and
%   minimum values, and a normalized product array with values in the
%   range [0, 1]. The input images must be of the same size.  They
%   can be of class uint8, unit16, or double. The outputs are of
%   class double.
%
%   Sample M-file used in Chapter 2.

fd = double(f);
gd = double(g);
p = fd.*gd;
pmax = max(p(:));
pmin = min(p(:));
pn = mat2gray(p);
```

```
function cr = imratio(f1, f2)
%IMRATIO Computes the ratio of the bytes in two images/variables.
%   CR = IMRATIO(F1, F2) returns the ratio of the number of bytes in
%   variables/files F1 and F2. If F1 and F2 are an original and
%   compressed image, respectively, CR is the compression ratio.
```

```
error(nargchk(2, 2, nargin));          % Check input arguments
cr = bytes(f1) / bytes(f2);            % Compute the ratio

%-----------------------------------------------------------------------%
function b = bytes(f)
% Return the number of bytes in input f. If f is a string, assume
% that it is an image filename; if not, it is an image variable.

if ischar(f)
   info = dir(f);         b = info.bytes;
elseif isstruct(f)
   % MATLAB's whos function reports an extra 124 bytes of memory
   % per structure field because of the way MATLAB stores
   % structures in memory.  Don't count this extra memory; instead,
   % add up the memory associated with each field.
   b = 0;
   fields = fieldnames(f);
   for k = 1:length(fields)
      elements = f.(fields{k});
      for m = 1:length(elements)
          b = b + bytes(elements(m));
      end
   end
else
   info = whos('f');      b = info.bytes;
end
```

```
function [X, R] = imstack2vectors(S, MASK)
%IMSTACK2VECTORS Extracts vectors from an image stack.
%   [X, R] = imstack2vectors(S, MASK) extracts vectors from S, which
%   is an M-by-N-by-n stack array of n registered images of size
%   M-by-N each (see Fig. 12.29). The extracted vectors are arranged
%   as the rows of array X. Input MASK is an M-by-N logical or
%   numeric image with nonzero values (1s if it is a logical array)
%   in the locations where elements of S are to be used in forming X
%   and 0s in locations to be ignored. The number of row vectors in
%   X is equal to the number of nonzero elements of MASK. If MASK is
%   omitted, all M*N locations are used in forming X.  A simple way
%   to obtain MASK interactively is to use function roipoly.
%   Finally, R is a column vector that contains the linear indices
%   of the locations of the vectors extracted from S.

% Preliminaries.
[M, N, n] = size(S);
if nargin == 1
   MASK = true(M, N);
else
   MASK = MASK ~= 0;
end
```

```
% Find the linear indices of the 1-valued elements in MASK. Each
% element of R identifies the location in the M-by-N array of the
% vector extracted from S.
R = find(MASK);

% Now find X.

% First reshape S into X by turning each set of n values along the
% third dimension of S so that it becomes a row of X. The order is
% from top to bottom along the first column, the second column, and
% so on.
Q = M*N;
X = reshape(S, Q, n);

% Now reshape MASK so that it corresponds to the right locations
% vertically along the elements of X.
MASK = reshape(MASK, Q, 1);

% Keep the rows of X at locations where MASK is not 0.
X = X(MASK, :);


function [x, y] = intline(x1, x2, y1, y2)
%INTLINE Integer-coordinate line drawing algorithm.
%  [X, Y] = INTLINE(X1, X2, Y1, Y2) computes an
%  approximation to the line segment joining (X1, Y1) and
%  (X2, Y2) with integer coordinates.  X1, X2, Y1, and Y2
%  should be integers.  INTLINE is reversible; that is,
%  INTLINE(X1, X2, Y1, Y2) produces the same results as
%  FLIPUD(INTLINE(X2, X1, Y2, Y1)).

dx = abs(x2 - x1);
dy = abs(y2 - y1);

% Check for degenerate case.
if ((dx == 0) && (dy == 0))
   x = x1;
   y = y1;
  return;
end

flip = 0;
if (dx >= dy)
   if (x1 > x2)
      % Always "draw" from left to right.
      t = x1; x1 = x2; x2 = t;
      t = y1; y1 = y2; y2 = t;
      flip = 1;
   end
```

```
    m = (y2 - y1)/(x2 - x1);
    x = (x1:x2).';
    y = round(y1 + m*(x - x1));
else
    if (y1 > y2)
        % Always "draw" from bottom to top.
        t = x1; x1 = x2; x2 = t;
        t = y1; y1 = y2; y2 = t;
        flip = 1;
    end
    m = (x2 - x1)/(y2 - y1);
    y = (y1:y2).';
    x = round(x1 + m*(y - y1));
end

if (flip)
    x = flipud(x);
    y = flipud(y);
end

function phi = invmoments(F)
%INVMOMENTS Compute invariant moments of image.
%   PHI = INVMOMENTS(F) computes the moment invariants of the image
%   F. PHI is a seven-element row vector containing the moment
%   invariants as defined in equations (11.3-17) through (11.3-23) of
%   Gonzalez and Woods, Digital Image Processing, 2nd Ed.
%
%   F must be a 2-D, real, nonsparse, numeric or logical matrix.

if (ndims(F) ~= 2) || issparse(F) || ~isreal(F) || ...
          ~(isnumeric(F) || islogical(F))
    error(['F must be a 2-D, real, nonsparse, numeric or logical' ...
          'matrix.']);
end
F = double(F);

phi = compute_phi(compute_eta(compute_m(F)));

%--------------------------------------------------------------------%
function m = compute_m(F)

[M, N] = size(F);
[x, y] = meshgrid(1:N, 1:M);

% Turn x, y, and F into column vectors to make the summations a bit
% easier to compute in the following.
x = x(:);
y = y(:);
F = F(:);

% DIP equation (11.3-12)
```

```
m.m00 = sum(F);
% Protect against divide-by-zero warnings.
if (m.m00 == 0)
   m.m00 = eps;
end
% The other central moments:
m.m10 = sum(x .* F);
m.m01 = sum(y .* F);
m.m11 = sum(x .* y .* F);
m.m20 = sum(x.^2 .* F);
m.m02 = sum(y.^2 .* F);
m.m30 = sum(x.^3 .* F);
m.m03 = sum(y.^3 .* F);
m.m12 = sum(x .* y.^2 .* F);
m.m21 = sum(x.^2 .* y .* F);

%--------------------------------------------------------------------%
function e = compute_eta(m)

% DIP equations (11.3-14) through (11.3-16).

xbar = m.m10 / m.m00;
ybar = m.m01 / m.m00;

e.eta11 = (m.m11 - ybar*m.m10) / m.m00^2;
e.eta20 = (m.m20 - xbar*m.m10) / m.m00^2;
e.eta02 = (m.m02 - ybar*m.m01) / m.m00^2;
e.eta30 = (m.m30 - 3 * xbar * m.m20 + 2 * xbar^2 * m.m10) / ...
           m.m00^2.5;
e.eta03 = (m.m03 - 3 * ybar * m.m02 + 2 * ybar^2 * m.m01) / ...
           m.m00^2.5;
e.eta21 = (m.m21 - 2 * xbar * m.m11 - ybar * m.m20 + ...
           2 * xbar^2 * m.m01) / m.m00^2.5;
e.eta12 = (m.m12 - 2 * ybar * m.m11 - xbar * m.m02 + ...
           2 * ybar^2 * m.m10) / m.m00^2.5;

%--------------------------------------------------------------------%
function phi = compute_phi(e)

% DIP equations (11.3-17) through (11.3-23).

phi(1) = e.eta20 + e.eta02;
phi(2) = (e.eta20 - e.eta02)^2 + 4*e.eta11^2;
phi(3) = (e.eta30 - 3*e.eta12)^2 + (3*e.eta21 - e.eta03)^2;
phi(4) = (e.eta30 + e.eta12)^2 + (e.eta21 + e.eta03)^2;
phi(5) = (e.eta30 - 3*e.eta12) * (e.eta30 + e.eta12) * ...
         ( (e.eta30 + e.eta12)^2 - 3*(e.eta21 + e.eta03)^2 ) + ...
         (3*e.eta21 - e.eta03) * (e.eta21 + e.eta03) * ...
         ( 3*(e.eta30 + e.eta12)^2 - (e.eta21 + e.eta03)^2 );
phi(6) = (e.eta20 - e.eta02) * ( (e.eta30 + e.eta12)^2 - ...
                                 (e.eta21 + e.eta03)^2 ) + ...
```

```
        4 * e.eta11 * (e.eta30 + e.eta12) * (e.eta21 + e.eta03);
phi(7) = (3*e.eta21 - e.eta03) * (e.eta30 + e.eta12) * ...
         ( (e.eta30 + e.eta12)^2 - 3*(e.eta21 + e.eta03)^2 ) + ...
         (3*e.eta12 - e.eta30) * (e.eta21 + e.eta03) * ...
         ( 3*(e.eta30 + e.eta12)^2 - (e.eta21 + e.eta03)^2 );
```

```
function E = iseven(A)
%ISEVEN Determines which elements of an array are even numbers.
%   E = ISEVEN(A) returns a logical array, E, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to even numbers
%   in A, and 0s (FALSE) elsewhere.

%   STEVE: Needs copyright text block. Ralph

E = 2*floor(A/2) == A;
```

```
function D = isodd(A)
%ISODD Determines which elements of an array are odd numbers.
%   D = ISODD(A) returns a logical array, D, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to odd numbers in
%   A, and 0s (FALSE) elsewhere.

D = 2*floor(A/2) ~= A;
```

M

```
function movie2tifs(m, file)
%MOVIE2TIFS Creates a multiframe TIFF file from a MATLAB movie.
%   MOVIE2TIFS(M, FILE) creates a multiframe TIFF file from the
%   specified MATLAB movie structure, M.

% Write the first frame of the movie to the multiframe TIFF.
imwrite(frame2im(m(1)), file, 'Compression', 'none', ...
    'WriteMode', 'overwrite');

% Read the remaining frames and append to the TIFF file.
for i = 2:length(m)
    imwrite(frame2im(m(i)), file, 'Compression', 'none', ...
        'WriteMode', 'append');

end
```

P

```
function I = percentile2i(h, P)
%PERCENTILE2I Computes an intensity value given a percentile.
%       I = PERCENTILE2I(H, P) Given a percentile, P, and a histogram,
%       H, this function computes an intensity, I, representing the
```

```
%       Pth percentile and returns the value in I. P must be in the
%       range [0, 1] and I is returned as a value in the range [0, 1]
%       also.
%
%   Example:
%
%   Suppose that h is a uniform histogram of an 8-bit image.  Typing
%
%       I = percentile2i(h, 0.5)
%
%   would output I = 0.5. To convert to the (integer) 8-bit range
%   [0, 255], we let I = floor(255*I).
%
%   See also function i2percentile.

% Check value of P.
if P < 0 || P > 1
   error('The percentile must be in the range [0, 1].')
end

% Normalized the histogram to unit area. If it is already normalized
% the following computation has no effect.
h = h/sum(h);

% Cumulative distribution.
C = cumsum(h);

% Calculations.
idx = find(C >= P, 1, 'first');
% Subtract 1 from idx because indexing starts at 1, but intensities
% start at 0. Also, normalize to the range [0, 1].
I = (idx - 1)/(numel(h) - 1);
```

```
function B = pixeldup(A, m, n)
%PIXELDUP Duplicates pixels of an image in both directions.
%   B = PIXELDUP(A, M, N) duplicates each pixel of A M times in the
%   vertical direction and N times in the horizontal direction.
%   Parameters M and N must be integers.  If N is not included, it
%   defaults to M.

% Check inputs.
if nargin < 2
   error('At least two inputs are required.');
end
if nargin == 2
   n = m;
end

% Generate a vector with elements 1:size(A, 1).
u = 1:size(A, 1);
```

```
% Duplicate each element of the vector m times.
m = round(m); % Protect against nonintegers.
u = u(ones(1, m), :);
u = u(:);

% Now repeat for the other direction.
v = 1:size(A, 2);
n = round(n);
v = v(ones(1, n), :);
v = v(:);
B = A(u, v);
```

```
function angles = polyangles(x, y)
%POLYANGLES Computes internal polygon angles.
%   ANGLES = POLYANGLES(X, Y) computes the interior angles (in
%   degrees) of an arbitrary polygon whose vertices are given in
%   [X, Y], ordered in a clockwise manner.  The program eliminates
%   duplicate adjacent rows in [X Y], except that the first row may
%   equal the last, so that the polygon is closed.

% Preliminaries.
[x y] = dupgone(x, y); % Eliminate duplicate vertices.
xy = [x(:) y(:)];
if isempty(xy)
   % No vertices!
   angles = zeros(0, 1);
   return;
end
if size(xy, 1) == 1 || ~isequal(xy(1, :), xy(end, :))
   % Close the polygon
   xy(end + 1, :) = xy(1, :);
end

% Precompute some quantities.
d = diff(xy, 1);
v1 = -d(1:end, :);
v2 = [d(2:end, :); d(1, :)];
v1_dot_v2 = sum(v1 .* v2, 2);
mag_v1 = sqrt(sum(v1.^2, 2));
mag_v2 = sqrt(sum(v2.^2, 2));

% Protect against nearly duplicate vertices; output angle will be 90
% degrees for such cases. The "real" further protects against
% possible small imaginary angle components in those cases.
mag_v1(~mag_v1) = eps;
mag_v2(~mag_v2) = eps;
angles = real(acos(v1_dot_v2 ./ mag_v1 ./ mag_v2) * 180 / pi);

% The first angle computed was for the second vertex, and the
% last was for the first vertex. Scroll one position down to
```

```
% make the last vertex be the first.
angles = circshift(angles, [1, 0]);

% Now determine if any vertices are concave and adjust the angles
% accordingly.
sgn = convex_angle_test(xy);

% Any element of sgn that's -1 indicates that the angle is
% concave. The corresponding angles have to be subtracted
% from 360.
I = find(sgn == -1);
angles(I) = 360 - angles(I);

%-------------------------------------------------------------------%
function sgn = convex_angle_test(xy)
%   The rows of array xy are ordered vertices of a polygon. If the
%   kth angle is convex (>0 and <= 180 degress) then sgn(k) =
%   1. Otherwise sgn(k) = -1. This function assumes that the first
%   vertex in the list is convex, and that no other vertex has a
%   smaller value of x-coordinate. These two conditions are true in
%   the first vertex generated by the MPP algorithm. Also the
%   vertices are assumed to be ordered in a clockwise sequence, and
%   there can be no duplicate vertices.
%
%   The test is based on the fact that every convex vertex is on the
%   positive side of the line passing through the two vertices
%   immediately following each vertex being considered.  If a vertex
%   is concave then it lies on the negative side of the line joining
%   the next two vertices. This property is true also if positive and
%   negative are interchanged in the preceding two sentences.

% It is assumed that the polygon is closed.  If not, close it.
if size(xy, 1) == 1 || ~isequal(xy(1, :), xy(end, :))
   xy(end + 1, :) = xy(1, :);
end

% Sign convention: sgn = 1 for convex vertices (i.e, interior angle
% > 0 and <= 180 degrees), sgn = -1 for concave vertices.

% Extreme points to be used in the following loop.  A 1 is appended
% to perform the inner (dot) product with w, which is 1-by-3 (see
% below).
L = 10^25;
top_left = [-L, -L, 1];
top_right = [-L, L, 1];
bottom_left = [L, -L, 1];
bottom_right = [L, L, 1];

sgn = 1; % The first vertex is known to be convex.
```

```
% Start following the vertices.
for k = 2:length(xy) - 1
   pfirst= xy(k - 1, :);
   psecond = xy(k, :); % This is the point tested for convexity.
   pthird = xy(k + 1, :);
   % Get the coefficients of the line (polygon edge) passing
   % through pfirst and psecond.
   w = polyedge(pfirst, psecond);

   % Establish the positive side of the line w1x + w2y + w3 = 0.
   % The positive side of the line should be in the right side of
   % the vector (psecond - pfirst).  deltax and deltay of this
   % vector give the direction of travel. This establishes which of
   % the extreme points (see above) should be on the + side. If that
   % point is on the negative side of the line, then w is replaced
   % by -w.

   deltax = psecond(:, 1) - pfirst(:, 1);
   deltay = psecond(:, 2) - pfirst(:, 2);
   if deltax == 0 && deltay == 0
      error('Data into convexity test is 0 or duplicated.')
   end
   if deltax <= 0  && deltay >= 0 %Bottom_right should be on + side.
      vector_product = dot(w, bottom_right); % Inner product.
      w = sign(vector_product)*w;
   elseif deltax <= 0 && deltay <= 0 %Top_right should be on + side.
      vector_product = dot(w, top_right);
      w = sign(vector_product)*w;
   elseif deltax >= 0 && deltay <= 0  %Top_left should be on + side.
      vector_product = dot(w, top_left);
      w = sign(vector_product)*w;
   else % deltax >= 0 & deltay >= 0, so bottom_left should be on +
        % side.
      vector_product = dot(w, bottom_left);
      w = sign(vector_product)*w;
   end
   % For the vertex at psecond to be convex, pthird has to be on the
   % positive side of the line.
   sgn(k) = 1;
   if (w(1)*pthird(:, 1) + w(2)*pthird(:, 2) + w(3)) < 0
      sgn(k) = -1;
   end
end

%-------------------------------------------------------------------%
function w = polyedge(p1, p2)
%   Outputs the coefficients of the line passing through p1 and
%   p2. The line is of the form w1x + w2y + w3 = 0.

x1 = p1(:, 1);  y1 = p1(:, 2);
```

```
x2 = p2(:, 1);  y2 = p2(:, 2);
if x1 == x2
   w2 = 0;
   w1 = -1/x1;
   w3 = 1;
elseif y1 ==y 2
   w1 = 0;
   w2 = -1/y1;
   w3 = 1;
elseif x1 == y1 && x2 == y2
   w1 = 1;
   w2 = 1;
   w3 = 0;
else
   w1 = (y1 - y2)/(x1*(y2 - y1) - y1*(x2 - x1) + eps);
   w2 = -w1*(x2 - x1)/(y2 - y1);
   w3 = 1;
end
w = [w1, w2, w3];

%--------------------------------------------------------------------%
function [xg, yg] = dupgone(x, y)
% Eliminates duplicate, adjacent rows in [x y], except that the
% first and last rows can be equal so that the polygon is closed.

xg = x;
yg = y;
if size(xg, 1) > 2
   I = find((x(1:end - 1, :) == x(2:end, :)) & ...
            (y(1:end - 1, :) == y(2:end, :)));
   xg(I) = [];
   yg(I) = [];
end

function flag = predicate(region)
%PREDICATE Evaluates a predicate for function splitmerge
%   FLAG = PREDICATE(REGION) evaluates a predicate for use in
%   function splitmerge for Example 11.14 in Digital Image
%   Processing Using MATLAB, 2nd edition. REGION is a subimage, and
%   FLAG is set to TRUE if the predicate evaluates to TRUE for
%   REGION; FLAG is set to FALSE otherwise.

% Compute the standard deviation and mean for the intensities of the
% pixels in REGION.
sd = std2(region);
m = mean2(region);

% Evaluate the predicate.
flag = (sd > 10) & (m > 0) & (m < 125);
```

R

```
function [xn, yn] = randvertex(x, y, npix)
%RANDVERTEX Adds random noise to the vertices of a polygon.
%   [XN, YN] = RANDVERTEX[X, Y, NPIX] adds uniformly distributed
%   noise to the coordinates of vertices of a polygon. The
%   coordinates of the vertices are input in X and Y, and NPIX is the
%   maximum number of pixel locations by which any pair (X(i), Y(i))
%   is allowed to deviate. For example, if NPIX = 1, the location of
%   any X(i) will not deviate by more than one pixel location in the
%   x-direction, and similarly for Y(i). Noise is added independently
%   to the two coordinates.

% Convert to columns.
x = x(:);
y = y(:);

% Preliminary calculations.
L = length(x);
xnoise = rand(L, 1);
ynoise = rand(L, 1);
xdev = npix*xnoise.*sign(xnoise - 0.5);
ydev = npix*ynoise.*sign(ynoise - 0.5);

% Add noise and round.
xn = round(x + xdev);
yn = round(y + ydev);

% All pixel locations must be no less than 1.
xn = max(xn, 1);
yn = max(yn, 1);
```

```
function H = recnotch(notch, mode, M, N, W, SV, SH)
%RECNOTCH Generates rectangular notch (axes) filters.
%   H = RECNOTCH(NOTCH, MODE, M, N, W, SV, SH) generates an M-by-N
%   notch filter consisting of symmetric pairs of rectangles of
%   width W placed on the vertical and horizontal axes of the
%   (centered) frequency rectangle. The vertical rectangles start at
%   +SV and -SV on the vertical axis and extend to both ends of the
%   axis. Horizontal rectangles similarly start at +SH and -SH and
%   extend to both ends of the axis. These values are with respect
%   to the origin of the axes of the centered frequency rectangle.
%   For example, specifying SV = 50 creates a rectangle of width W
%   that starts 50 pixels above the center of the vertical axis and
%   extends up to the first row of the filter. A similar rectangle
%   is created starting 50 pixels below the center and extending to
%   the last row. W must be an odd number to preserve the symmetry
%   of the filtered Fourier transform.
%
%       Valid values of NOTCH are:
```

```
%
%           'reject'    Notchreject filter.
%
%           'pass'      Notchpass filter.
%
%
%       Valid values of MODE are:
%
%           'both'          Filtering on both axes.
%
%           'horizontal'    Filtering on horizontal axis only.
%
%           'vertical'      Filtering on vertical axis only.
%
%       One of these three values must be specified in the call.
%
%   H = RECNOTCH(NOTCH, MODE, M, N) sets W = 1, and SV = SH = 1.
%
%  H is of floating point class single. It is returned uncentered
%  for consistency with filtering function dftfilt. To view H as an
%  image or mesh plot, it should be centered using Hc = fftshift(H).

% Preliminaries.
if nargin == 4
   W = 1;
   SV = 1;
   SH = 1;
elseif nargin ~= 7
   error('The number of inputs must be 4 or 7.')
end
% AV and AH are rectangle amplitude values for the vertical and
% horizontal rectangles: 0 for notchreject and 1 for notchpass.
% Filters are computed initially as reject filters and then changed
% to pass if so specified in NOTCH.
if strcmp(mode, 'both')
   AV = 0;
   AH = 0;
elseif strcmp(mode, 'horizontal')
   AV = 1; % No reject filtering along vertical axis.
   AH = 0;
elseif strcmp(mode, 'vertical')
   AV = 0;
   AH = 1; % No reject filtering along horizontal axis.
end
if iseven(W)
   error('W must be an odd number.')
end

% Begin filter computation. The filter is generated as a reject
% filter. At the end, it are changed to a notchpass filter if it
```

```
% is so specified in parameter NOTCH.
H = rectangleReject(M, N, W, SV, SH, AV, AH);

% Finished computing the rectangle notch filter. Format the
% output.
H = processOutput(notch, H);

%-----------------------------------------------------------------%
function H = rectangleReject(M, N, W, SV, SH, AV, AH)
% Preliminaries.
H = ones(M, N, 'single');
% Center of frequency rectangle.
UC = floor(M/2) + 1;
VC = floor(N/2) + 1;
% Width limits.
WL = (W - 1)/2;
% Compute rectangle notches with respect to center.
% Left, horizontal rectangle.
H(UC-WL:UC+WL, 1:VC-SH) = AH;
% Right, horizontal rectangle.
H(UC-WL:UC+WL, VC+SH:N) = AH;
% Top vertical rectangle.
H(1:UC-SV, VC-WL:VC+WL) = AV;
% Bottom vertical rectangle.
H(UC+SV:M, VC-WL:VC+WL) = AV;

%-----------------------------------------------------------------%
function H = processOutput(notch, H)
% Uncenter the filter to make it compatible with other filters in
% the DIPUM toolbox.
H = ifftshift(H);
% Generate a pass filter if one was specified.
if strcmp(notch, 'pass')
   H = 1 - H;
end
```

S

```
function seq2tifs(s, file)
%SEQ2TIFS Creates a multi-frame TIFF file from a MATLAB sequence.

% Write the first frame of the sequence to the multiframe TIFF.
imwrite(s(:, :, :, 1), file, 'Compression', 'none', ...
    'WriteMode', 'overwrite');

% Read the remaining frames and append to the TIFF file.
for i = 2:size(s, 4)
    imwrite(s(:, :, :, i), file, 'Compression', 'none', ...
        'WriteMode', 'append');
end
```

```
function v = showmo(cv, i)
%SHOWMO Displays the motion vectors of a compressed image sequence.
%   SHOWMO(CV, I) displayes the motion vectors for frame I of a
%   TIFS2CV compressed sequence of images.
%
%   See also TIFS2CV and CV2TIFS.

frms = double(cv.frames);
m = double(cv.blksz);
q = double(cv.quality);

if q == 0
    ref = double(huff2mat(cv.video(1)));
else
    ref = double(jpeg2im(cv.video(1)));
end

fsz = size(ref);
mvsz = [fsz/m 2 frms];
mv = int16(huff2mat(cv.motion));
mv = reshape(mv, mvsz);
v = zeros(fsz, 'uint8') + 128;

% Create motion vector image.
for j = 1:mvsz(1) * mvsz(2)

    x1 = 1 + mod(m * (j - 1), fsz(1));
    y1 = 1 + m * floor((j - 1) * m / fsz(1));

    x2 = x1 - mv(1 + floor((x1 - 1) / m), ...
          1 + floor((y1 - 1) / m), 1, i);
    y2 = y1 - mv(1 + floor((x1 - 1) / m), ...
          1 + floor((y1 - 1) / m), 2, i);

    [x, y] = intline(x1, double(x2), y1, double(y2));
    for k = 1:length(x) - 1
        v(x(k), y(k)) = 255;
    end
    v(x(end), y(end)) = 0;
end

imshow(v);

function [dist, angle] = signature(b, x0, y0)
%SIGNATURE Computes the signature of a boundary.
%   [DIST, ANGLE, XC, YC] = SIGNATURE(B, X0, Y0) computes the
%   signature of a given boundary. A signature is defined as the
%   distance from (X0, Y0) to the boundary, as a function of angle
%   (ANGLE). B is an np-by-2 array (np > 2) containing the (x, y)
%   coordinates of the boundary ordered in a clockwise or
```

```
%   counterclockwise direction. If (XO, YO) is not included in the
%   input argument, the centroid of the boundary is used by default.
%   The maximum size of arrays DIST and ANGLE is 360-by-1,
%   indicating a maximum resolution of one degree. The input must be
%   a one-pixel-thick boundary obtained, for example, by using
%   function bwboundaries.
%
%   If (XO, YO) or the default centroid is outside the boundary, the
%   signature is not defined and an error is issued.

% Check dimensions of b.
[np, nc] = size(b);
if (np < nc || nc ~= 2)
   error('b must be of size np-by-2.');
end

% Some boundary tracing programs, such as boundaries.m, result in a
% sequence in which the coordinates of the first and last points are
% the same. If this is the case, in b, eliminate the last point.
if isequal(b(1, :), b(np, :))
   b = b(1:np - 1, :);
   np = np - 1;
end

% Compute the origin of vector as the centroid, or use the two
% values specified. Use the same symbol (xc, yc) in case the user
% includes (xc, yc) in the output call.
if nargin == 1
   xO = sum(b(:, 1))/np; % Coordinates of the centroid.
   yO = sum(b(:, 2))/np;
end

% Check to see that (xc, yc) is inside the boundary.
IN = inpolygon(xO, yO, b(:, 1), b(:, 2));
if ~IN
   error('(xO, yO) or centroid is not inside the boundary.')
end

% Shift origin of coordinate system to (xO, yO).
b(:, 1) = b(:, 1) - xO;
b(:, 2) = b(:, 2) - yO;

% Convert the coordinates to polar.  But first have to convert the
% given image coordinates, (x, y), to the coordinate system used by
% MATLAB for conversion between Cartesian and polar cordinates.
% Designate these coordinates by (xcart, ycart). The two coordinate
% systems are related as follows:  xcart = y and ycart = -x.
xcart = b(:, 2);
ycart = -b(:, 1);
[theta, rho] = cart2pol(xcart, ycart);
```

```
% Convert angles to degrees.
theta = theta.*(180/pi);

% Convert to all nonnegative angles.
j = theta == 0; % Store the indices of theta = 0 for use below.
theta = theta.*(0.5*abs(1 + sign(theta)))...
        - 0.5*(-1 + sign(theta)).*(360 + theta);
theta(j) = 0; % To preserve the 0 values.

% Round theta to 1 degree increments.
theta = round(theta);

% Keep theta and rho together for sorting purposes.
tr = [theta, rho];

% Delete duplicate angles.  The unique operation also sorts the
% input in ascending order.
[w, u] = unique(tr(:, 1));
tr = tr(u,:); % u identifies the rows kept by unique.

% If the last angle equals 360 degrees plus the first angle, delete
% the last angle.
if tr(end, 1) == tr(1) + 360
   tr = tr(1:end - 1, :);
end

% Output the angle values.
angle = tr(:, 1);

% Output the length values.
dist = tr(:, 2);
```

```
function [srad, sang, S] = specxture(f)
%SPECXTURE Computes spectral texture of an image.
%   [SRAD, SANG, S] = SPECXTURE(F) computes SRAD, the spectral energy
%   distribution as a function of radius from the center of the
%   spectrum, SANG, the spectral energy distribution as a function of
%   angle for 0 to 180 degrees in increments of 1 degree, and S =
%   log(1 + spectrum of f), normalized to the range [0, 1]. The
%   maximum value of radius is min(M,N), where M and N are the number
%   of rows and columns of image (region) f. Thus, SRAD is a row
%   vector of length = (min(M, N)/2) - 1; and SANG is a row vector of
%   length 180.

% Obtain the centered spectrum, S, of f. The variables of S are
% (u, v), running from 1:M and 1:N, with the center (zero frequency)
% at [M/2 + 1, N/2 + 1] (see Chapter 4).
S = fftshift(fft2(f));
```

```
S = abs(S);
[M, N] = size(S);
x0 = M/2 + 1;
y0 = N/2 + 1;

% Maximum radius that guarantees a circle centered at (x0, y0) that
% does not exceed the boundaries of S.
rmax = min(M, N)/2 - 1;

% Compute srad.
srad = zeros(1, rmax);
srad(1) = S(x0, y0);
for r = 2:rmax
   [xc, yc] = halfcircle(r, x0, y0);
   srad(r) = sum(S(sub2ind(size(S), xc, yc)));
end

% Compute sang.
[xc, yc] = halfcircle(rmax, x0, y0);
sang = zeros(1, length(xc));
for a = 1:length(xc)
   [xr, yr] = radial(x0, y0, xc(a), yc(a));
   sang(a) = sum(S(sub2ind(size(S), xr, yr)));
end

% Output the log of the spectrum for easier viewing, scaled to the
% range [0, 1].
S = mat2gray(log(1 + S));

%--------------------------------------------------------------------%
function [xc, yc] = halfcircle(r, x0, y0)
%   Computes the integer coordinates of a half circle of radius r and
%   center at (x0,y0) using one degree increments.
%
%   Goes from 91 to 270 because we want the half circle to be in the
%   region defined by top right and top left quadrants, in the
%   standard image coordinates.

theta=91:270;
theta = theta*pi/180;
[xc, yc] = pol2cart(theta, r);
xc = round(xc)' + x0; % Column vector.
yc = round(yc)' + y0;

%--------------------------------------------------------------------%
function [xr, yr] = radial(x0, y0, x, y)
%   Computes the coordinates of a straight line segment extending
%   from (x0, y0) to (x, y).
%
%   Based on function intline.m.  xr and yr are returned as column
%   vectors.
```

```
[xr, yr] = intline(x0, x, y0, y);

function [v, unv] = statmoments(p,  n)
%STATMOMENTS Computes statistical central moments of image histogram.
%   [W, UNV] = STATMOMENTS(P, N) computes up to the Nth statistical
%   central moment of a histogram whose components are in vector
%   P. The length of P must equal 256 or 65536.
%
%   The program outputs a vector V with V(1) = mean, V(2) = variance,
%   V(3) = 3rd moment, . . . V(N) = Nth central moment. The random
%   variable values are normalized to the range [0, 1], so all
%   moments also are in this range.
%
%   The program also outputs a vector UNV containing the same moments
%   as V, but using un-normalized random variable values (e.g., 0 to
%   255 if length(P) = 2^8). For example, if length(P) = 256 and V(1)
%   = 0.5, then UNV(1) would have the value UNV(1) = 127.5 (half of
%   the [0 255] range).

Lp = length(p);
if (Lp ~= 256) && (Lp ~= 65536)
   error('P must be a 256- or 65536-element vector.');
end
G = Lp - 1;

% Make sure the histogram has unit area, and convert it to a
% column vector.
p = p/sum(p); p = p(:);

% Form a vector of all the possible values of the
% random variable.
z = 0:G;

% Now normalize the z's to the range [0, 1].
z = z./G;

% The mean.
m = z*p;

% Center random variables about the mean.
z = z - m;

% Compute the central moments.
v = zeros(1, n);
v(1) = m;
for j = 2:n
   v(j) = (z.^j)*p;
end

if nargout > 1
```

```
   % Compute the uncentralized moments.
   unv = zeros(1, n);
   unv(1)=m.*G;
   for j = 2:n
      unv(j) = ((z*G).^j)*p;
   end
end

function t = statxture(f, scale)
%STATXTURE Computes statistical measures of texture in an image.
%   T = STATXURE(F, SCALE) computes six measures of texture from an
%   image (region) F. Parameter SCALE is a 6-dim row vector whose
%   elements multiply the 6 corresponding elements of T for scaling
%   purposes. If SCALE is not provided it defaults to all 1s.  The
%   output T is 6-by-1 vector with the following elements:
%     T(1) = Average gray level
%     T(2) = Average contrast
%     T(3) = Measure of smoothness
%     T(4) = Third moment
%     T(5) = Measure of uniformity
%     T(6) = Entropy

if nargin == 1
   scale(1:6) = 1;
else % Make sure it's a row vector.
   scale = scale(:)';
end

% Obtain histogram and normalize it.
p = imhist(f);
p = p./numel(f);
L = length(p);

% Compute the three moments. We need the unnormalized ones
% from function statmoments. These are in vector mu.
[v, mu] = statmoments(p, 3);

% Compute the six texture measures:
% Average gray level.
t(1) = mu(1);
% Standard deviation.
t(2) = mu(2).^0.5;
% Smoothness.
% First normalize the variance to [0 1] by
% dividing it by (L - 1)^2.
varn = mu(2)/(L - 1)^2;
t(3) = 1 - 1/(1 + varn);
% Third moment (normalized by (L - 1)^2 also).
t(4) = mu(3)/(L - 1)^2;
% Uniformity.
t(5) = sum(p.^2);
```

```
% Entropy.
t(6) = -sum(p.*(log2(p + eps)));

% Scale the values.
t = t.*scale;

function s = subim(f, m, n, rx, cy)
%SUBIM Extract subimage.
%   S = SUBIM(F, M, N, RX, CY) extracts a subimage, S, from the input
%   image, F.  The subimage is of size M-by-N, and the coordinates of
%   its top, left corner are (RX, CY).
%
%   Sample M-file used in Chapter 2.

s = zeros(m, n);
rowhigh = rx + m - 1;
colhigh = cy + n - 1;
xcount = 0;
for r = rx:rowhigh
   xcount = xcount + 1;
   ycount = 0;
   for c = cy:colhigh
      ycount = ycount + 1;
      s(xcount, ycount) = f(r, c);
   end
end
```

T

```
function m = tifs2movie(file)
%TIFS2MOVIE Create a MATLAB movie from a multiframe TIFF file.
%   M = TIFS2MOVIE(FILE) creates a MATLAB movie structure from a
%   multiframe TIFF file.

% Get file info like number of frames in the multi-frame TIFF
info = imfinfo(file);
frames = size(info, 1);

% Create a gray scale map for the UINT8 images in the MATLAB movie
gmap = linspace(0, 1, 256);
gmap = [gmap' gmap' gmap'];

% Read the TIFF frames and add to a MATLAB movie structure.
for i = 1:frames
    [f, fmap] = imread(file, i);
    if (strcmp(info(i).ColorType, 'grayscale'))
        map = gmap;
    else
        map = fmap;
    end
```

```
    m(i) = im2frame(f, map);
end

function s = tifs2seq(file)
%TIFS2SEQ Create a MATLAB sequence from a multi-frame TIFF file.

% Get the number of frames in the multi-frame TIFF.
frames = size(imfinfo(file), 1);

% Read the first frame, preallocate the sequence, and put the first
% in it.
i = imread(file, 1);
s = zeros([size(i) 1 frames], 'uint8');
s(:,:,:,1) = i;

% Read the remaining TIFF frames and add to the sequence.
for i = 2:frames
    s(:,:,:,i) = imread(file, i);
end

function [out, revertclass] = tofloat(in)
%TOFLOAT Convert image to floating point
%   [OUT, REVERTCLASS] = TOFLOAT(IN) converts the input image IN to
%   floating-point. If IN is a double or single image, then OUT
%   equals IN.  Otherwise, OUT equals IM2SINGLE(IN).  REVERTCLASS is
%   a function handle that can be used to convert back to the class
%   of IN.

identity = @(x) x;
tosingle = @im2single;

table = {'uint8',   tosingle, @im2uint8
   'uint16',  tosingle, @im2uint16
   'int16',   tosingle, @im2int16
   'logical', tosingle, @logical
   'double',  identity, identity
   'single',  identity, identity};

classIndex = find(strcmp(class(in), table(:, 1)));

if isempty(classIndex)
   error('Unsupported input image class.');
end

out = table{classIndex, 2}(in);

revertclass = table{classIndex, 3};

function [rt, f, g] = twodsin(A, u0, v0, M, N)
%TWODSIN Compare for-loops vs. vectorization.
```

```
%   The comparison is based on implementing the function f(x, y) =
%   Asin(u0x + v0y) for x = 0, 1, 2,..., M - 1 and y = 0, 1, 2,...,
%   N - 1. The inputs to the function are M and N and the constants
%   in the function.
%
%   Sample M-file used in Chapter 2.

% First implement using for loops.

tic   % Start timing.

for r = 1:M
   u0x = u0*(r - 1);
   for c = 1:N
      v0y = v0*(c - 1);
      f(r, c) = A*sin(u0x + v0y);
   end
end

t1 = toc;   % End timing.

%      Now implement using vectorization.  Call the image g.

tic   % Start timing.

r = 0:M - 1;
c = 0:N - 1;
[C, R] = meshgrid(c, r);
g = A*sin(u0*R + v0*C);

t2 = toc;   % End timing.

%      Compute the ratio of the two times.

rt = t1/(t2 + eps); % Use eps in case t2 is close to 0
```

W

```
function w = wave2gray(c, s, scale, border)
%WAVE2GRAY Display wavelet decomposition coefficients.
%   W = WAVE2GRAY(C, S, SCALE, BORDER) displays and returns a
%   wavelet coefficient image.
%
%   EXAMPLES:
%     wave2gray(c, s);                      Display w/defaults.
%     foo = wave2gray(c, s);                Display and return.
%     foo = wave2gray(c, s, 4);             Magnify the details.
%     foo = wave2gray(c, s, -4);            Magnify absolute values.
%     foo = wave2gray(c, s, 1, 'append');   Keep border values.
```

```
%
%   INPUTS/OUTPUTS:
%     [C, S] is a wavelet decomposition vector and bookkeeping
%     matrix.
%
%     SCALE       Detail coefficient scaling
%     -------------------------------------------------------
%     0 or 1      Maximum range (default)
%     2,3...      Magnify default by the scale factor
%     -1, -2...   Magnify absolute values by abs(scale)
%
%     BORDER      Border between wavelet decompositions
%     -------------------------------------------------------
%     'absorb'    Border replaces image (default)
%     'append'    Border increases width of image
%
%     Image W:   ------- ------ ------------ ------------------
%                |      |      |            |
%                | a(n) | h(n) |            |
%                |      |      |            |
%                ------- ------     h(n-1)   |
%                |      |      |            |
%                | v(n) | d(n) |            |        h(n-2)
%                |      |      |            |
%                ------- ------ ------------
%                |             |            |
%                |    v(n-1)   |   d(n-1)   |
%                |             |            |
%                -------------- ------------ ------------------
%                |                          |
%                |          v(n-2)          |        d(n-2)
%                |                          |
%
%     Here, n denotes the decomposition step scale and a, h, v, d are
%     approximation, horizontal, vertical, and diagonal detail
%     coefficients, respectively.

% Check input arguments for reasonableness.
error(nargchk(2, 4, nargin));

if (ndims(c) ~= 2) || (size(c, 1) ~= 1)
  error('C must be a row vector.');   end

if (ndims(s) ~= 2) || ~isreal(s) || ~isnumeric(s) || (size(s,2) ~= 2)
  error('S must be a real, numeric two-column array.');   end

elements = prod(s, 2);
if (length(c) < elements(end)) || ...
      ~(elements(1) + 3 * sum(elements(2:end - 1)) >= elements(end))
   error(['[C S] must be a standard wavelet ' ...
```

```
        'decomposition structure.']);
end

if (nargin > 2) && (~isreal(scale) || ~isnumeric(scale))
   error('SCALE must be a real, numeric scalar.');
end

if (nargin > 3) && (~ischar(border))
  error('BORDER must be character string.');
end

if nargin == 2
   scale = 1;  % Default scale.
end

if nargin < 4
   border = 'absorb';  % Default border.
end

% Scale coefficients and determine pad fill.
absflag = scale < 0;
scale = abs(scale);
if scale == 0
   scale = 1;
end

[cd, w] = wavecut('a', c, s);   w = mat2gray(w);
cdx = max(abs(cd(:))) / scale;
if absflag
   cd = mat2gray(abs(cd), [0, cdx]);   fill = 0;
else
   cd = mat2gray(cd, [-cdx, cdx]);   fill = 0.5;
end

% Build gray image one decomposition at a time.
for i = size(s, 1) - 2:-1:1
   ws = size(w);

   h = wavecopy('h', cd, s, i);
   pad = ws - size(h);     frontporch = round(pad / 2);
   h = padarray(h, frontporch, fill, 'pre');
   h = padarray(h, pad - frontporch, fill, 'post');

   v = wavecopy('v', cd, s, i);
   pad = ws - size(v);     frontporch = round(pad / 2);
   v = padarray(v, frontporch, fill, 'pre');
   v = padarray(v, pad - frontporch, fill, 'post');

   d = wavecopy('d', cd, s, i);
   pad = ws - size(d);     frontporch = round(pad / 2);
```

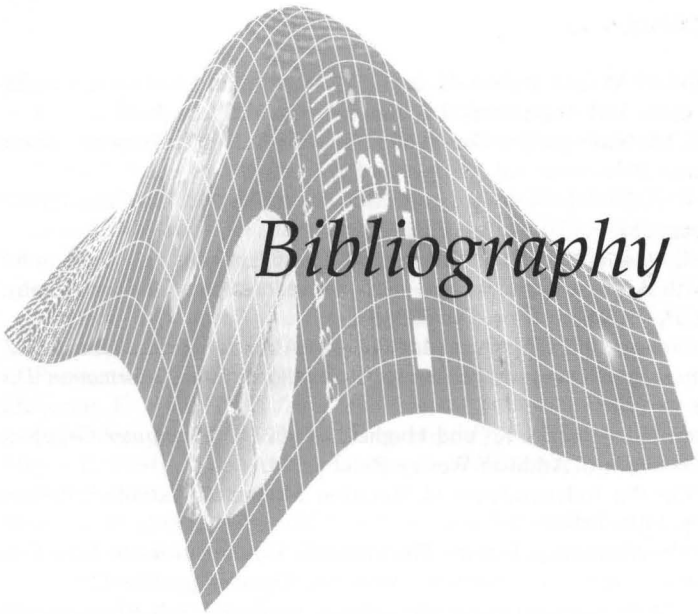
```
   d = padarray(d, frontporch, fill, 'pre');
   d = padarray(d, pad - frontporch, fill, 'post');

   % Add 1 pixel white border.
   switch lower(border)
   case 'append'
      w = padarray(w, [1 1], 1, 'post');
      h = padarray(h, [1 0], 1, 'post');
      v = padarray(v, [0 1], 1, 'post');
   case 'absorb'
      w(:, end) = 1;   w(end, :) = 1;
      h(end, :) = 1;   v(:, end) = 1;
   otherwise
      error('Unrecognized BORDER parameter.');
   end

   w = [w h; v d];                  % Concatenate coefs.
end

if nargout == 0
   imshow(w);                       % Display result.
end
}
```

X

```
function [C, theta] = x2majoraxis(A, B)
%X2MAJORAXIS Aligns coordinate x with the major axis of a region.
%   [C, THETA] = X2MAJORAXIS(A, B) aligns the x-coordinate
%   axis with the major axis of a region or boundary. The y-axis is
%   perpendicular to the x-axis.  The rows of 2-by-2 matrix A are
%   the coordinates of the two end points of the major axis, in the
%   form A = [x1 y1; x2 y2]. Input B is either a binary image (i.e.,
%   an array of class logical) containing a single region, or it is
%   an np-by-2 set of points representing a (connected) boundary. In
%   the latter case, the first column of B must represent
%   x-coordinates and the second column must represent the
%   corresponding y-coordinates. Output C contains the same data as
%   the input, but aligned with the major axis. If the input is an
%   image, so is the output; similarly the output is a sequence of
%   coordinates if the input is such a sequence. Parameter THETA is
%   the initial angle between the major axis and the x-axis. The
%   origin of the xy-axis system is at the bottom left; the x-axis
%   is the horizontal axis and the y-axis is the vertical.
%
%   Keep in mind that rotations can introduce round-off errors when
%   the data are converted to integer (pixel) coordinates, which
%   typically is a requirement.  Thus, postprocessing (e.g., with
%   bwmorph) of the output may be required to reconnect a boundary.
```

```
% Preliminaries.
if islogical(B)
   type = 'region';
elseif size(B, 2) == 2
   type = 'boundary';
   [M, N] = size(B);
   if M < N
      error('B is boundary. It must be of size np-by-2; np > 2.')
   end
   % Compute centroid for later use. c is a 1-by-2 vector.
   % Its 1st component is the mean of the boundary in the x-direction.
   % The second is the mean in the y-direction.
   c(1) = round((min(B(:, 1)) + max(B(:, 1))/2));
   c(2) = round((min(B(:, 2)) + max(B(:, 2))/2));

   % It is possible for a connected boundary to develop small breaks
   % after rotation. To prevent this, the input boundary is filled,
   % processed as a region, and then the boundary is re-extracted.
   % This guarantees that the output will be a connected boundary.
   m = max(size(B));
   % The following image is of size m-by-m to make sure that there
   % there will be no size truncation after rotation.
   B = bound2im(B,m,m);
   B = imfill(B,'holes');
else
   error('Input must be a boundary or a binary image.')
end

% Major axis in vector form.
v(1) = A(2, 1) - A(1, 1);
v(2) = A(2, 2) - A(1, 2);
v = v(:);  % v is a col vector

% Unit vector along x-axis.
u = [1; 0];

% Find angle between major axis and x-axis. The angle is
% given by acos of the inner product of u and v divided by
% the product of their norms. Because the inputs are image
% points, they are in the first quadrant.
nv = norm(v);
nu = norm(u);
theta = acos(u'*v/nv*nu);
if theta > pi/2
   theta = -(theta - pi/2);
end
theta = theta*180/pi;  % Convert angle to degrees.

% Rotate by angle theta and crop the rotated image to original size.
C = imrotate(B, theta, 'bilinear', 'crop');
```

```
% If the input was a boundary, re-extract it.
if  strcmp(type, 'boundary')
   C = boundaries(C);
   C = C{1};
   % Shift so that centroid of the extracted boundary is
   % approx equal to the centroid of the original boundary:
   C(:, 1) = C(:, 1) - min(C(:, 1)) + c(1);
   C(:, 2) = C(:, 2) - min(C(:, 2)) + c(2);
end
```



Bibliography

References Applicable to All Chapters:

Gonzalez, R. C. and Woods, R. E. [2008]. *Digital Image Processing*, 3rd ed., Prentice Hall, Upper Saddle River, NJ.

Hanselman, D. and Littlefield, B. R. [2005]. *Mastering MATLAB 7*, Prentice Hall, Upper Saddle River, NJ.

Image Processing Toolbox, Users Guide, Version 6.2. [2008], The MathWorks, Inc., Natick, MA.

Using MATLAB, Version 7.7 [2008], The MathWorks, Inc., Natick, MA

Other References Cited:

Acklam, P. J. [2002]. "MATLAB Array Manipulation Tips and Tricks." Available for download at http://home.online.no/~pjacklam/matlab/doc/mtt/ and also from the Tutorials section at www.imageprocessingplace.com.

Bell, E.T, [1965]. Men of Mathematics, Simon & Schuster, NY.

Brigham, E. O. [1988]. *The Fast Fourier Transform and its Applications*, Prentice Hall, Upper Saddle River, NJ.

Bribiesca, E. [1981]. "Arithmetic Operations Among Shapes Using Shape Numbers," *Pattern Recog.*, vol. 13, no. 2, pp. 123–138.

Bribiesca, E., and Guzman, A. [1980]. "How to Describe Pure Form and How to Measure Differences in Shape Using Shape Numbers," *Pattern Recog.*, vol. 12, no. 2, pp. 101–112.

Brown, L. G. [1992]. "A Survey of Image Registration Techniques," *ACM Computing Surveys*, vol. 24, pp. 325–376.

Canny, J. [1986]. "A Computational Approach for Edge Detection," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 8, no. 6, pp. 679–698.

CIE [2004]. *CIE 15:2004. Technical Report: Colorimetry*, 3rd ed. (can be obtained from www.techstreet.com/ciegate.tmpl)

Dempster, A. P., Laird, N. M., and Ruben, D. B. [1977]. "Maximum Likelihood from Incomplete Data via the EM Algorithm," *J. R. Stat. Soc. B*, vol. 39, pp. 1–37.

Di Zenzo, S. [1986]. "A Note on the Gradient of a Multi-Image," *Computer Vision, Graphics and Image Processing*, vol. 33, pp. 116–125.

Eng, H.-L. and Ma, K.-K. [2001]. "Noise Adaptive Soft-Switching Median Filter," IEEE Trans. Image Processing, vol.10, no. 2, pp. 242–251.

Fischler, M. A. and Bolles, R. C. [1981]. "Random Sample Consensus: A Paradigm for Model Fitting with Application to Image Analysis and Automated Cartography," *Comm. of the ACM*, vol. 24, no. 6, pp. 381–395.

Floyd, R. W. and Steinberg, L. [1975]. "An Adaptive Algorithm for Spatial Gray Scale," *International Symposium Digest of Technical Papers, Society for Information Displays*, 1975, p. 36.

Foley, J. D., van Dam, A., Feiner S. K., and Hughes, J. F. [1995]. *Computer Graphics: Principles and Practice in C*, Addison-Wesley, Reading, MA.

Flusser, J. [2000]. "On the Independence of Rotation Moment Invariants," Pattern Recog., vol. 33, pp. 1405–1410.

Gardner, M. [1970]. "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'," *Scientific American*, October, pp. 120–123.

Gardner, M. [1971]. "Mathematical Games On Cellular Automata, Self-Reproduction, the Garden of Eden, and the Game 'Life'," *Scientific American*, February, pp. 112–117.

Goshtasby, A. A. [2005]. *2-D and 3-D Image Registration*, Wiley Press., NY

Hanisch, R. J., White, R. L., and Gilliland, R. L. [1997]. "Deconvolution of Hubble Space Telescope Images and Spectra," in *Deconvolution of Images and Spectra*, P. A. Jansson, ed., Academic Press, NY, pp. 310–360.

Haralick, R. M. and Shapiro, L. G. [1992]. *Computer and Robot Vision*, vols. 1 & 2, Addison-Wesley, Reading, MA.

Harris, C. and Stephens, M. [1988]. "A Combined Corner and Edge Detector," *Proc. 4th Alvey Vision Conference*, pp. 147–151.

Holmes, T. J. [1992]. "Blind Deconvolution of Quantum-Limited Incoherent Imagery," *J. Opt. Soc. Am. A*, vol. 9, pp. 1052–1061.

Holmes, T. J., et al. [1995]. "Light Microscopy Images Reconstructed by Maximum Likelihood Deconvolution," in *Handbook of Biological and Confocal Microscopy*, 2nd ed., J. B. Pawley, ed., Plenum Press, NY, pp. 389–402.

Hough, P.V.C. [1962]. "Methods and Means for Recognizing Complex Patterns." U.S. Patent 3,069,654.

Hu, M. K. [1962]. "Visual Pattern Recognition by Moment Invariants," *IRE Trans. Info. Theory*, vol. IT-8, pp. 179–187.

ICC [2004]. *Specification ICC.1:2004-10 (Profile version 4.2.0.0): Image Technology Colour Management—Architecture, Profile Format, and Data Structure*, International Color Consortium.

ISO [2004]. *ISO 22028-1:2004(E). Photography and Graphic Technology—Extended Colour Encodings for Digital Image Storage, Manipulation and Interchange. Part 1: Architecture and Requirements*. (Can be obtained from www.iso.org.)

Jansson, P. A., ed. [1997]. *Deconvolution of Images and Spectra*, Academic Press, NY.

Keys, R. G. [1983]. "Cubic Convolution Interpolation for Digital Image Processing," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-29, no. 6, pp. 1153–1160.

Kim, C. E. and Sklansky, J. [1982]. "Digital and Cellular Convexity," *Pattern Recog.*, vol. 15, no. 5, pp. 359–367.

Klete, R. and Rosenfeld, A. [2004]. Digital Geometry—Geometric Methods for Digital Picture Analysis, Morgan Kaufmann, San Francisco.

Leon-Garcia, A. [1994]. *Probability and Random Processes for Electrical Engineering*, 2nd. ed., Addison-Wesley, Reading, MA.

Lucy, L. B. [1974]. "An Iterative Technique for the Rectification of Observed Distributions," *The Astronomical Journal*, vol. 79, no. 6, pp. 745–754.

Mamistvalov, A. [1998]. "*n*-Dimensional Moment Invariants and Conceptual Mathematical Theory of Recognition [of] *n*-Dimensional Solids," *IEEE Trans. Pattern Anal. Machine Intell.*, vol.20, no. 8. pp. 819–831.

McNames, J. [2006]. "An Effective Color Scale for Simultaneous Color and Gray-scale Publications," *IEEE Signal Processing Magazine*, vol. 23, no. 1, pp. 82–96.

Meijering, E. H. W. [2002]. "A Chronology of Interpolation: From Ancient Astronomy to Modern Signal and Image Processing," *Proc. IEEE*, vol. 90, no. 3, pp. 319–342.

Meyer, F. [1994] . "Topographic Distance and Watershed Lines," *Signal Processing*, vol. 38, pp. 113-125.

Moravec, H. [1980]. "Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover," *Tech. Report CMU-RI-TR-3*, Carnegie Mellon University, Robotics Institute, Pittsburgh, PA.

Morovic, J. [2008]. *Color Gamut Mapping*, Wiley, NY.

Noble, B. and Daniel, J. W. [1988]. *Applied Linear Algebra*, 3rd ed., Prentice Hall, Upper Saddle River, NJ.

Otsu, N. [1979] "A Threshold Selection Method from Gray-Level Histograms," *IEEE Trans. Systems, Man, and Cybernetics*, vol. SMC-9, no. 1, pp. 62–66.

Peebles, P. Z. [1993]. *Probability, Random Variables, and Random Signal Principles*, 3rd ed., McGraw-Hill, NY.

Prince, J. L. and Links, J. M. [2006]. *Medical Imaging Signals and Systems*, Prentice Hall, Upper Saddle River, NJ.

Poynton, C. A. [1996]. *A Technical Introduction to Digital Video*, Wiley, NY.

Ramachandran, G. N. and Lakshminarayanan, A. V. [1971]. "Three-Dimensional Reconstruction from Radiographs and Electron Micrographs: Applications of Convolution instead of Fourier Transforms," *Proc. Natl. Aca. Sc.*, vol. 68, pp. 2236–2240.

Richardson, W. H. [1972]. "Bayesian-Based Iterative Method of Image Restoration," *J. Opt. Soc. Am.*, vol. 62, no. 1, pp. 55–59.

Rogers, D. F. [1997]. *Procedural Elements of Computer Graphics*, 2nd ed., McGraw-Hill, NY.

Russ, J. C. [2007]. *The Image Processing Handbook*, 4th ed., CRC Press, Boca Raton, FL.

Sharma, G. [2003]. *Digital Color Imaging Handbook*, CRC Press, Boca Raton, FL.

Shep, L. A. and Logan, B. F. [1974]. "The Fourier Reconstruction of a Head Section," *IEEE Trans. Nuclear Sci.*, vol. NS-21, pp. 21–43.

Shi, J. amd Tomasi, C. [1994]. "Good Features to Track," *IEEE Conf. Computer Vision and Pattern Recognition* (CVPR94), pp. 593–600.

Sklansky, J., Chazin, R. L., and Hansen, B. J. [1972]. "Minimum-Perimeter Polygons of Digitized Silhouettes," *IEEE Trans. Computers*, vol. C-21, no. 3, pp. 260–268.

Sloboda, F., Zatko, B., and Stoer, J. [1998]. "On Approximation of Planar One-Dimensional Continua," in *Advances in Digital and Computational Geometry*, R. Klette, A. Rosenfeld, and F. Sloboda (eds.), Springer, Singapore, pp. 113–160.

Soille, P. [2003]. *Morphological Image Analysis: Principles and Applications*, 2nd ed., Springer-Verlag, NY.

Stokes, M., Anderson, M., Chandrasekar, S., and Motta, R. [1996]. "A Standard Default Color Space for the Internet—sRGB," available at http://www.w3.org/Graphics/Color/sRGB..

Sze, T. W. and Yang, Y. H. [1981]. " A Simple Contour Matching Algorithm," I*EEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-3, no. 6, pp. 676–678.

Szeliski, R. [2006]. "Image Alignment and Stitching: A Tutorial," *Foundations and Trends in Computer Graphics and Vision*, vol. 2, no. 1, pp. 1–104.

Trucco, E. and Verri, A. [1998]. *Introductory Techniques for 3-D Computer Vision*, Prentice Hall, Upper Saddle River, NJ.

Ulichney, R. [1987], *Digital Halftoning*, The MIT Press, Cambridge, MA.

Hu, M. K. [1962]. "Visual Pattern Recognition by Moment Invariants," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 179–187.

Van Trees, H. L. [1968]. Detection, Estimation, and Modulation Theory, Part I, Wiley, NY.

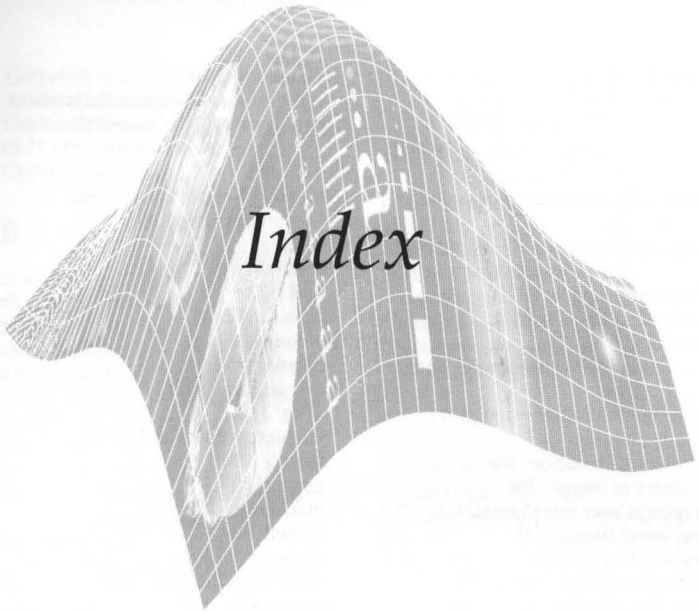
Vincent, L. [1993], "Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms, " *IEEE Trans. on Image Processing* vol. 2, no. 2, pp. 176–201.

Vincent, L. and Soille, P. [1991]. "Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations, " *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 13, no. 6, pp. 583–598.

Wolbert, G. [1990]. *Digital Image Warping*, IEEE Computer Society Press, Los Alamitos, CA.

Zadeh, L. A. [1965]. "Fuzzy Sets," *Inform and Control*, vol. 8, pp. 338–353.

Zitová B. and Flusser J. [2003]. "Image Registration Methods: A Survey," *Image and Vision Computing*, vol. 21, no. 11, pp. 977–1000.



Index

Symbols

4-connectivity 515
8-connectivity 515
: (colon in MATLAB) 33
. (dot) 46
... (dots for long equations) 24
.mat. *See* MAT-file
@ operator 63
>> (prompt) 8
; (semicolon in MATLAB) 16

A

abs 168
adapthisteq 108
Adjacency 515
adpmedian 235
aggfcn 149
AND 53
  elementwise 53
  scalar 53
angle 171
annotation 102
ans 55
appcoef2 398
applycform 344
applylut 507
approxfcn 152
Arctangent 165, 367
  four quadrant 165, 367. *See also* atan2
Array 15, 42. *See also* Matrix
  operations 47
  preallocating 65
  selecting dimension 42
  standard 43
  vs. matrix 15
atan2 170
Autocorrelation 682
Average image power 241
axis 96
axis ij (moves axis origin) 96
axis off 191
axis on 191
axis xy (moves axis origin) 96

B

Background 489, 498, 509, 514, 557, 498
  nonuniform 527, 532, 549, 558, 571
bandfilter 199
bar 95
bayesgauss 685
bellmf 145, 157
Binary image. *See* Image
bin2dec 438
Bit depth. *See* Color image processing
blanks 692
Blind deconvolution. *See* Image restoration
blkproc 459
Book web site 7
Border. *See* Boundary, Region
bound2eight 605
bound2four 605
bound2im 600
Boundaries
  functions for extracting 598
Boundary 598. *See also* Region
  axis (major, minor) 626
  basic rectangle 626
  changing direction of 599
  connecting 605
  defined 598
  diameter 626
  eccentricity 626
  length 625
  minimally connected 598
  minimum-perimeter polygons 610
  ordering a random sequence of boundary points 605
  segments 622
break 58, 61
bsubsamp 605
bsxfun 676
bwboundaries 599
bwdist 589
bwhitmiss 505
bwlabel 515
bwmorph 511
bwperim 598

C

cart2pol 621
Cartesian product 487
Cassini spacecraft 206
cat 319
CDF. See Cumulative distribution function
ceil 171
cell 392, 431
Cell arrays 74
  example 76
celldisp 75, 431
cellfun 75
cellplot 431
cellstr 692
Cellular complex 612

Center of frequency rectangle 171
Center of mass 516, 643
cform structure 344
Chain codes. *See* Representation and description
`char` 26, 73
`checkerboard` 238
`circshift` 604
Circular convolution. *See* Convolution
Classes. *See also* Image classes
  converting between 28
  list 26
  terminology 28
Classification. *See* Recognition
`clc` 9
`clear` 9
Clipping 30
C MEX-file 442
`cnotch` 202
Code. *See also* Function, Programming
  combining statements 32
  long lines 24
  modular 216
  optimization 65
  preallocation 65
  vectorization 68
`col2im` 460
`colfilt` 118
`colon` 33
`colorgrad` 369
Colon notation. *See* Notation
Color image processing
  basics of 349
  bit depth 318
  brightness 340
  chromaticity 340
  chromaticity diagram 341
  CIE 341
  color balancing 358
  color correction 358
  color edge detection 366
  color editing 352
  color gamut 347
  color image segmentation 372
  color map 321
  color map matrix 321
  color maps 324
  color profile 331, 346
  color space
    CMY 330
    CMYK 330
    device independent 340
    HSI 331
    HSV 329
    L*a*b* 344
    L*ch 344
    NTSC 328
    RGB 319
    sRGB 343
    u´v´L 344
    uvL 344
    xyY 341
    XYZ 341
    YCbCr 329
  color transformations 350
  converting between CIE and sRGB 344
  converting between color spaces 328
  converting between RGB, indexed, and gray-scale images. 324
  converting HSI to RGB 334
  converting RGB to HSI 334
  dithering 323, 326
  extracting RGB component images 319
  full-color transformation 351
  gamut mapping 347
  gradient of a vector 368
  gradient of image 366
  graphical user interface (GUI) 353
  gray-level slicing 325
  gray-scale map 321
  histogram equalization 359
  hue 328, 332, 340
  ICC color profiles 346, 347
  image sharpening 365
  image smoothing 360
  indexed images 321
  intensity 332
  luminance 320
  line of purples 342
  manipulating RGB and indexed images 323
  perceptual uniformity 343
  primaries of light 319
  pseudocolor mapping 351
  RGB color cube 319
  RGB color image 318
  RGB values of colors 322
  saturation 328, 332, 340
  secondaries of light 319
  shade 329
  soft proofing 347
  spatial filtering 360
  tint 329
  tone 329
  trichromatic coefficients 340
  tristimulus values 340
`colormap` 191, 323
`colorseg` 373
Column vector. *See* Vector
Command-function duality 24
`compare` 423
`computer` 55
Conjugate transpose 33
Connected
  component 515, 597
  pixels 597
  set 598
`connectpoly` 605
`continue` 58, 62
Contour. *See* Boundary
Contrast
  enhancement. *See* Image enhancement
  measure of, 667
  stretching. *See* Image enhancement
Control points. See Geometric transformations
`conv2` 393
converting between linear and subscript 40
Convex
  deficiency 622
  hull 622
  vertex 612
Convolution
  circular 174
  expression 114, 244
  filter 110
  frequency domain 173
  kernel 110
  mask 110
  mechanics 110
  spatial 80
  theorem 173
Convolution theorem 173
`conwaylaws` 509
Co-occurrence matrix. *See* Representation and description
  image 14
  MATLAB 14
Coordinates 14
  Cartesian 192, 621
  image 13
  pixel 14
  polar 257, 621, 654
  row and column 14
  spatial 14
`copper` 323
Corner 633
Corner detection. *See* Representation and description
`cornermetric` 638
`cornerprocess` 638
Correlation 114, 681
  coefficient 312, 682
  expression 114
  mechanics 110
  normalized cross-correlation 312
  spatial 110
  theorem 242
Correlation coefficient. *See* Correlation
Covariance matrix 684
  approximation 662
  function for computing 663
`covmatrix` 663
`cpselect` 306
Cross-correlation 312, 682. *See also* Recognition
CT 251
`cumsum` 101
Cumulative distribution function 99, 212
  transformation 99
  table of 214
Current directory. See MATLAB

Curvature. *See* Representation and description
Custom function 2, 7
`cv2tifs` 483
Cygnus Loop 587

D

dc component 165
`dec2base` 700
`dec2bin` 436, 446
`deconvblind` 250
`deconvlucy` 247
Deconvolution. *See* Image restoration
`deconvreg` 245
`deconvwnr` 241
`defuzzify` 149
Descriptor. *See* Representation and description
`detcoef2` 398
DFT. *See* Discrete Fourier transform
`dftfilt` 179
`dftuv` 186
`diag` 374
`diameter` 626
`diff` 529
Digital image. *See* Image
Digital image processing, definition 3
Dimension
  array 16
  singleton 17
Directory 16
Discrete cosine transform (DCT) 456
Discrete Fourier transform (DFT)
  centering 167, 168
  computing 168
  defined 164, 165
  filtering. *See* Frequency domain filtering
  inverse 165
  periodicity 166
  phase angle 165
  power spectrum 166
  scaling issues 172
  spectrum 165
  visualizing 168
  wraparound error 174
`disp` 71
Displacement variable 114
Distance 372
  computing in MATLAB 675
  Euclidean 343, 372, 675
  Mahalanobis 373, 678, 684
  transform 589
`dither` 323
Division by zero 47
`doc` 10
Don't care pixel 506
Dots per inch. *See* Dpi
`double` 26
Dpi 24
`dwtmode` 387

E

`edge` 542
Edge detection. *See* Image segmentation
`edgetaper` 242
`edit` 46
`eig` 665
Eigenvalues 637, 663
  for corner detection 637
Electromagnetic spectrum 2
Elementwise operation. *See* Operation.
`else` 58
`elseif` 58
`end` 34
End point 507
`endpoints` 507
Entropy 645, 651
`eps` 55
`error` 59
`eval` 694
Extended minima transform 595
`eye` 44

F

Faceted shading 193
`false` 44, 587
False contouring 23
`fan2para` 274
`fanbeam` 269
Fast wavelet transform (FWT) 380
`fchcode` 607
Features 306, 625, 674. *See also* Representation and description
`fft2` 168
`fftshift` 169
Fields. *See* Structures
`figure` 19
`filter` 575
Filter(ing)
  frequency domain. *See* Frequency domain filtering
  morphological. *See* Morphology
  spatial. *See* Spatial filtering
`find` 215
`fix` 152
`fliplr` 262
`flipud` 262
Floating point number. *See* Number
`floor` 171
`for` 58, 59
Foreground 489, 490, 503, 507, 557, 598
`format` 56
Fourier
  coefficients 165
  descriptors 627
  Slice theorem 257
  spectrum 165
  transform. *See* Discrete Fourier transform (DFT)
`fplot` 98, 156
`frdescp` 629
Freeman chain codes. *See* Representation and description
Frequency
  domain 165
  convolution 173
  rectangle 165
  rectangle center 171
  variables 165
Frequency domain filtering
  bandpass 199
  bandreject 199
  basic steps 178
  constrained least squares 244
  convolution 173
  direct inverse 240
  fundamentals 173
  high-frequency emphasis 197
  highpass 194
  lowpass 187
  M-function for 179
  notchpass 202
  notchreject 202
  periodic noise reduction 236
  steps 178
  Wiener 240
Frequency domain filters. *See also* Frequency domain filtering
  bandpass 199
  bandreject 199
  Butterworth bandreject 199
  Butterworth highpass 195
  Butterworth lowpass 187
  constrained least squares 244
  converting to spatial filters 181
  direct inverse 240
  from spatial filters 180
  Gaussian highpass 195
  Gaussian lowpass 188
  generating directly 185
  high-frequency emphasis 197
  highpass 194
  ideal bandreject 199
  ideal highpass 195
  ideal lowpass 187
  notchreject 202
  padding 174
  periodic noise reduction 236
  plotting 190
  pseudoinverse. *See* Image restoration
  Ram-Lak 259, 266
  sharpening 194
  Shepp-Logan 259
  smoothing 187
  transfer function 173
  Wiener 240
  zero-phase-shift 179
`freqz2` 181

fspecial 120
full 43
Function
  body 45
  comments 45
  custom 7
  decision 679
  discriminant 679
  factories 141
  function-generating 141
  H1 line 45
  handle 63, *66*, 119
    anonymous 64
    named 63
    simple 63
  help text 45
  M-file 4, 10
    components of 45
  M-function 4, 44
  nested, 140
  programming. *See* Programming
  subfunction 45
  windowing. *See* Windowing functions
  wrapper 298
fuzzyfilt 162
Fuzzy processing
  aggregation 135, 138
  aggregation, function for 149
  custom membership functions 143
  definitions 129
  defuzzification 136, 138
  defuzzification, function for 149
  degree of membership 129
  ELSE rule 139
  fuzzification 133
  fuzzy set 129
  general model 139
  IF-THEN rules 133
    antecedent 133
    conclusion 133
    consequent 133
    firing level 139
    premise 133
    strength level 139
  implication 134, 137
  implication, function for 147
  improving performance 151
  inference 134
  intensity transformations 155
  lambda functions 146
  linguistic value 133
  linguistic variable 133
  logical operations 137
  membership function 129, 131
  overall system function 150
  rule strength, function for 146
  spatial filtering 158
  universe of discourse 129
  using fuzzy sets 133
fuzzysysfcn 150

G

Gaussian bandreject 199
gca 96
Generalized delta functions. *See* Image reconstruction
Geometric transformations
  affine transformations 283
    affine matrix 284
    similarity transformations 285
  applying to images 288
  control points 306, 351
  controlling the output grid 297
  forward transformation (mapping) 278
  global transformations 306
  homogeneous coordinates 284
  horizon line 288
  image coordinate systems 291
  input space 278
  interpolation 299
    1-D 299
    2-D 302
    bicubic 302
    bilinear 302
    comparing methods 302
    cubic 302
  kernels 300
  linear 301
    nearest-neighbor 302
    resampling 300
  local transformations 306
  inverse transformation (mapping) 279, 288
  output image location 293
  output space 278
  shape-preserving 285
  projective transformations 287
  tiles 107
  vanishing points 288
get 56, 353
getsequence 496
global 430
Gradient
  defined 366
  morphological 524
  used for edge detection. *See* Image segmentation
Graphical user interface (GUI) 353
gray2ind 325
graycomatrix 648
graycoprops 649
Gray level. *See also* Intensity
  definition 2, 13, 27
  transformation function 81
grayslice 325
graythresh 562
grid off 191
grid on 191
gscale 92

H

H1 line 45
Handle. *See* Function handle
help 46
hilb 39
Hilbert matrix 39
hist 220
histc 437
histeq 100
Histogram. *See also* Image enhancement
  bimodal 558
  contrast-limited 107
  defined 94
  equalization 99
  equalization of color images 359
  matching 102
  normalized 94
  plotting 94
  specification 102
  unimodal 558
histroi 227
hold on 98
Hole. *See also* Morphology, Region
  definition 598
  filling 520
Hotelling transform 662
hough 553
Hough transform. *See also* Image segmentation
  accumulator cells 552
  functions for computing 552
  line detection 556
  line linking 556
  parameter space 551
houghlines 555
houghpeaks 555
hpfilter 195
hsi2rgb 338
hsv2rgb 330
huff2mat 440
huffman 429
hypot 187
Hysteresis thresholding. *See* Image segmentation

I

i 55
i2percentile 567
ICC. *See* International Color Consortium
  color profiles 346
iccread 347
ice 352
Icon notation. *See also* Notation
  custom function 7
  MATLAB Wavelet Toolbox 377
  Image Processing Toolbox 7
IDFT. *See* Inverse discrete Fourier transform
if 58
IF-THEN rule. *See* Fuzzy processing

ifanbeam 272
ifft2 172
ifftshift 170
ifrdescp 629
Illumination bias 575
im2bw 29, 31
im2col 460
im2double 29
im2frame 473
im2jpeg 457
im2jpeg2k 466
im2minperpoly 617
im2single 29
im2uint8 29
im2uint16 29
imadjust 82
imag 170
Image 2
  amplitude 2
  analysis 3
  as a matrix 15
  average power 241
  binary 27, 598
  classes 26
    converting between 23
  columns 14
  coordinates 13
  definition 2
  description. *See* Representation and description
  digital 2, 14
  displaying 18
  dithering 323
  element 2, 15
  format extensions 17
  formats 17
  gray level. *See* Gray level, Intensity
  gray-scale 27
  indexed 27
  intensity. *See* Intensity
  interpolation. *See* Geometric transformations
  monochrome 13
  multispectral 666, 686
  origin 14
  padding 110, 118, 174
  picture element 2
  representation. *See* Representation and description
  resolution 24
  RGB 13, 27
  rows 14
  size 14
  spatial coordinates 2
  Tool 19
  types 27
  understanding 3
  writing 21
Image compression
  background 421
  coding redundancy 424
  compression ratio 421
  decoder 421
  encoder 421
  error free 423
  Huffman 427
    code 427
      block code 428
      decodable 428
      instantaneous 428
    codes 427
    decoding 439
    encoding 433
  improved gray-scale (IGS) quantization 453
  information preserving 423
  inverse mapper 424
  irrelevant infomation 453
  JPEG 2000 compression 464
    coding system 464
    subbands 464
  JPEG compression
    discrete cosine transform (DCT) 456
    JPEG standard 456
  lossless 423
  lossless predictive coding 449
  predictor 449
  quantization 453
  quantizer 424
  reversible mappings 449
  rms 423
  root mean square error 423
  spatial redundancy 446
    interpixel redundancy 448
  symbol coder 424
  symbol decode 424
  video compression 472
    image sequences in MATLAB 473
    motion compensation 476
    movies in MATLAB 473
    multiframe TIFF files 472
    temporal redundancy 472, 476
    video frames 472
Image enhancement 80, 164
  color. *See* Color image processing
  contrast enhancement, stretching 84, 85, 90, 529
  frequency domain filtering 164
    high-frequency emphasis 197
    periodic noise removal 204
    sharpening 194
    smoothing 188
  histogram
    adaptive equalization 107
    equalization 99
    matching (specification) 102
    processing 93
  intensity transformations 81
    arbitrary 86
    contrast-stretching 84
    functions for computing 82, 89
    logarithmic 84
  spatial filtering
    geometric mean 119
    noise reduction 127
    sharpening 120
    smoothing (blurring) 116
    using fuzzy sets 155
Image Processing Toolbox 1, 4, 7
Image reconstruction
  absorption profile 252
  background 252
  backprojection 253, 259
  center ray 268
  computed tomography 251
  fan-beam 259
  fan-beam data 268
  filter implementation 258
  filtered projection 258
  Fourier slice theorem 257
  generalized delta functions 258
  parallel-ray beam 255
  Radon transform 254
  Ram-Lak filter 259, 266
  ray sum 254
  Shepp-Logan filter 259
  Shepp-Logan head phantom 261
  sinogram 263
  slice 254, 257
  windowing functions. *See* Windowing functions
Image registration
  area-based 311
  automatic registration 316
  basic process 306
  control points 306
  correlation coefficient 312
  feature detector 316
  inferring transformation parameters 307
  inliers 317
  manual feature selection 306
  manual matching 306
  mosaicking 316
  normalized cross-correlation 312
  outliers 317
  similarity metrics 314
Image restoration
  adaptive spatial filters 233. *See also* Spatial filters
  deconvolution 210
    blind, 237, 250
  direct inverse filtering 240
  iterative 247
  linear 210
  Lucy-Richardson algorithm 246
  model 210
  noise models 211. *See also* Noise
  noise only 229
  nonlinear 247
  constrained least squares filtering 244
  optical transfer function 210
  parametric Wiener filter 241
  periodic noise reduction 236
  point spread function 210
  pseudoinverse 240
  spatial noise filters. *See also* Spatial filters
  regularized filtering 244

Wiener filtering 240
Image segmentation
edge detection 541
Canny detector 546
double edges 542, 546
gradient angle 541
gradient magnitude 541
gradient vector 541
Laplacian 542
Laplacian of a Gaussian (LoG) detector 545
location 542
masks 544
Prewitt detector 543, 545
Roberts detector 543, 545
Sobel detector 542
using function edge 541
zero crossings 543
zero-crossings detector 546
image thresholding
using local statistics 571
line detection 538
masks 538
using the Hough transform 549
nonmaximal suppression 546
oversegmentation 591
point detection 536
region-based 578
logical predicate 578
region growing 578
region splitting and merging 582
edge map 549
thresholding 557
background point 557
basic global thresholding 559
hysteresis 546
local statistics 571
object (foreground) point 557
Otsu's (optimum) method 561
separability measure 562
types of 558
using edges 567
using image smoothing 565
using moving averages 575
using watersheds 588
catchment basin 588
marker-controlled 593
using gradients 591
using the distance transform 589
watershed 588
watershed transform 588
Image Tool 19
imapprox 321
imbothat 529
imclearborder 521
imclose 501
imcomplement 83
imdilate 492
imerode 500
imextendedmin 595
imfilter 114
imfill 521, 603
imfinfo 23
imhist 94, 156
imhmin 531
imimposemin 595
imlincomb 50
imnoise 126, 211
imnoise2 216
imnoise3 221
imopen 501
implay 407, 474
implfcns 147
imratio 421
imread 15
imreconstruct 518
imregionalmin 593
imrotate 291, 659
imshow 18, 69
imstack2vectors 663
imtool 19
imtophat 529
imtransform 288
imtransform2 298
imwrite 21
ind2gray 325
ind2rgb 326
ind2sub 40
Indexing 33
linear 39
logical 38
matrix 35
row-column 40
single colon 37
subscript 33
vector 33
Inf 47
InitialMagnification 510
inpolygon 616
input 72
int2str 699
int8 26
int16 26
int32 26
Intensity. *See also* Gray level
definition 2, 13, 27
scaling 92
transformation function 81
arbitrary 86
contrast-stretching 84
fuzzy 155
histogram. *See* Histogram
logarithmic 84
thresholding 85
utility M-functions 87
transformations 80
International Color Consortium 346
Interpolation. *See* Geometric transformations
interp1 86
interp1q 351
interpn 153
intline 606
intrans 89, 157
invmoments 658
iptsetpref 291
iradon 263
iscell 54
iscellstr 54, 694
ischar 54
isempty 54
isequal 54
iseven 203
isfield 54
isfinite 54
isinf 54
isinteger 54
isletter 54
islogical 28, 54
ismember 54
isnan 54
isnumeric 54
isodd 203
isprime 54
isreal 54
isscalar 54
isspace 54
issparse 54
isstruct 54
isvector 54
Inverse discrete Fourier transform 165

J

j 55
jpeg2im 461
jpeg2k2im 468
JPEG compression 456

L

Label matrix 515
lambdafcns 146
Laplacian
defined 120
mask for 121, 122
of a Gaussian (LoG). *See* Image segmentation
of color images 365
of vectors 365
used for edge detection. *See* Image segmentation
Laplacian of a Gaussian (LoG) 545
LaTeX-style notation 553
length 59
Line
detection. *See* Image segmentation, Hough transform
linking. *See* Hough transform
normal representation 551
slope-intercept representation 551
Line detection. *See* Image segmentation
linspace 34, 157
load 11
localmean 572
localthresh 573
log 84
log2 84
log10 84
logical 26, 27

Logical
array 27
class 27
indexing 38, 216
mask 125, 225, 587
operator 52
long 57
Long lines. *See* Code
long e 57
long eng 57
long g 57
lookfor 46
Lookup table 87, 506
lower 201
lpc2mat 451
lpfilter 189
Lucy-Richardson algorithm. *See* Image restoration

M

magic 44
Magic square 44
mahalanobis 678
makecform 344
makecounter 141
makefuzzyedgesys 161
makelut 507
maketform 279
Mammogram 83
manualhist 105
Marker image 518, 567, 584, 593. *See also* Morphology
Mask. *See* Logical mask, Spatial mask, Morphological reconstruction
mat2gray 29, 30
mat2huff 436
mat2str 699
Matching. *See* Recognition
MAT-file 11
MATLAB 1, 2
background 4
command history 9
command window 8
coordinate convention 14
current directory 8
current directory field 8
definition 4
desktop 7
desktop tools 9
editor/debugger 10
function factories 141
function-generating functions 141
function plotting 93
help 10
help browser 10
image coordinate systems 291
M-file. *See* Function
M-function. *See* Function
nested functions . *See* Function
plotting 190
prompt 16
retrieving work 11
saving work 11
search path 9
string operations 692
toolboxes 4
workspace 8
workspace browser 8
Matrix
as an image 15
interval. *See* Morphology
operations 47
sparse 42
vs. array 15
Matrix vs. array 15
max 48, 686
Maximum likelihood 250
mean 76, 517
mean2 76, 92
Mean vector 684
approximation 662
function for computing 663
medfilt2 126
Median 126. *See also* Spatial filtering, Spatial filters
mesh 190
meshgrid 69
Metacharacters 695
mexErrMsgTxt 445
MEX-file 442
min 48
Minima imposition 595
Minimum-perimeter polygons 610, 703. See also Representation and description
Moiré pattern 203
Moment(s)
about the mean 224
central 224
invariants 656
statistical 632
used for texture analysis 644
Monospace characters 15
montage 474
Morphology, Morphological
4-connectivity 515
8-connectivity 515
closing 500
combining dilation and erosion 500
connected component 514
definition 515
labeling 514
label matrix 515
dilation 490
erosion 497
filtering 503, 524, 526
gradient 524
gray-scale morphology
alternating sequential filtering 526
bottomhat transformation 529
close-open filtering 526
closing 524
dilation 521
erosion 521
granulometry 529
open-close filtering 526
opening 524
reconstruction 530
closing-by-reconstruction 531
h-minima transform 531
opening-by-reconstruction 531
tophat-by-reconstruction 532
surface area 529
tophat transformation 528
hit-or-miss transformation 503
interval matrix 506
lookup table 506
matching 503
opening 500
pruning 512
parasitic components 512
reconstruction 518
clearing border objects 521
filling holes 520
mask 518
marker 518
opening by reconstruction 518
reflection of set 488
shrinking 512
skeleton 511
spurs 512
structuring element 486, 490
decomposition 493
flat 522
origin 488, 491, 492
strel function 494
thickening 512
thinning 511
translation of set 488
view of binary images 489
Mosaicking 316
movie2avi 475
movingthresh 576
movie2tifs 475
MPP. *See* Minimum-perimeter polygons
mxArray 445
mxCalloc 445
mxCreate 445
mxGet 445

N

NaN 47, 55
nargchk 88
nargin 87
nargout 87
ndims 42
Neighborhood processing 80, 109
Nested function. *See* Function
nextpow2 175
nlfilt 117
Noise
adding 211
application areas 213
average power 241
density 215

Erlang 214
parameter
estimating 224
scaling 211
exponential 214
filters. *See* Filter(ing)
gamma. *See* Erlang above
Gaussian 214
lognormal 214
models 211
multiplicative 211
periodic 220
Poisson 211, 247
Rayleigh 212, 214
salt and pepper 214, 215
speckle 211
uniform 214
with specified distribution 212
Noise-to-signal power ratio 241
`norm` 675
Norm. *See* Vector norm
Normalized cross-correlation. *See* Correlation
`normxcorr2` 313, 682
NOT 53
Notation
colon 33
function listing 7
icon 7
LaTeX-style 553
`ntrop` 426
`ntsc2rgb` 329
Number
exponential notation 56
floating point 55
format types 57
precision 55
representation 55
`numel` 59

O

Object recognition. *See* Recognition
`onemf` 146
`ones` 44
Operation
array 47
elementwise 47
matrix 47
Operator
arithmetic 46
logical 52
relational 50
OR 53
elementwise 53
scalar 53
`ordfilt2` 125
Ordering boundary points 605
OTF (optical transfer function) 210
`otf2psf` 210
`otsuthresh` 564

P

`padarray` 118
`paddedsize` 174
Padding. *See* Image padding
Panning 604
`para2fan` 275
`patch` 320
Pattern recognition. *See* Recognition
PDF. *See* probability density function
Pel 2, 15. *See also* Pixel
Percentile 567
`percentile2i` 567
`permute` 677
`persistent` 507
`phantom` 261
`pi` 55
Picture element 2, 15
Pixel
coordinates 14
definition 2, 15
`pixeldup` 238
Pixel(s)
adjacent 515
connected 515, 597
connecting 605
ordering along a boundary 605
path 515
straight digital line between two points 606
Pixels(s)
orientation of triplets 612
`plot` 41, 98
Plotting 93, 98
surface 190
wireframe 190
Point detection. *See* Image segmentation
`pointgrid` 282
`pol2cart` 621
`polyangles` 704
Polymersome cells 563
`pow2` 438
Preallocating arrays 65. *See also* Code
Predicate
function 585
logical 578
Predicate (logical) 578
Principal components
for data compression 667
for object alignment 670
transform 662
`principalcomps` 664
`print` 26
Probability. *See also* Histogram
density function 99
for equalization 99
specified 103
table of 214
of intensity level 94
`prod` 119
Programming. *See also* Code, Function
`break` 61
code optimization 65
commenting code 45
`continue` 58, 61
floating-point numbers 55
flow control 57
function body 45
function definition line 45
H1 line 45
help text 45
if construct 58
interactive I/O 71
M-Function 44
loops 59, 60
number formats 57
operators 46
`switch` 62
values 55
variable number of inputs and outputs 87
vectorizing 68
wrapper function 298
Prompt 8
PSF (point spread function) 210
`psf2otf` 210

Q

`qtdecomp` 584
`qtgetblk` 584
`quad` 64
Quadimages 583
Quadregions 583
Quadtree 583
Quantization 14
`quantize` 454

R

`radon` 260
Radon transform 254
`rand` 44, 215
`randn` 44, 215
Random
variable 211, 224
number generator 213
`randvertex` 704
RANSAC 316
`real` 170
`realmax` 55
`realmin` 55
Recognition
decision boundary 679
decision function 679
decision-theoretic methods 679
adaptive learning systems 691
Bayes classifier 684
correlation 681
correlation template 681
minimum-distance classifiers 680
discriminant function 679
distance measures 675
feature 674

hyperplane 681
matching. See also Cross-correlation
correlation 681
minimum-distance 680
morphological. *See* Morphology
template 312, 681
pattern 674
pattern class 674
pattern vector 674, 680
structural methods 691
regular expressions 694
string matching 693, 701
string registration 701, 704
working with pattern strings in MATLAB 692
`reflect` 492
`regexp` 695
`regexpi` 696
`regexprep` 696
Region
adjacent 578
background points 598
border 598
boundary 598
contour 598
functions for extracting 598
interior point 59, 598
of interest 225
Regional descriptors. *See* Representation and description
`regiongrow` 580
Region growing. *See* Image segmentation
Region merging. *See* Image segmentation
`regionprops` 642
Region splitting. *See* Image segmentation
Regular expressions 694
`rem` 152, 392
Representation and description
background 597
description approaches 625
boundary descriptors 625
axis (major, minor) 626
basic rectangle 626
corners 633
curvature 703
diameter 626
Fourier descriptors 627
length 625
shape numbers 626
statistical moments 632
regional descriptors
co-occurrence matrices 647
function `regionprops` 642
moment invariants 656
principal components 661
texture 644
region and boundary extraction 598
representation approaches
boundary segments 622
chain codes 606
Freeman chain codes 606
normalizing 606
minimum-perimeter polygons 610, 703
normalizing chain codes 606
signatures 619
`reprotate` 303
Resampling 300
`reshape` 401, 438
Resolution. *See* Image
`return` 58
`rgb2gray` 326
`rgb2hsi` 337
`rgb2hsv` 330
`rgb2ind` 325
`rgb2ntsc` 328
`rgb2ycbcr` 329
`rgbcube` 320
Ringing 187, 242
ROI. *See* Region of interest
`roipoly` 225
`rot90` 115
`round` 25
Row vector. *See* Vector

S

Sampling
definition 14
`save` 11
Scalar 15
Scripts 44
Scrolling 604
`seq2tifs` 475
`set` 96
Set
element 128
fuzzy. *See* Fuzzy processing
theory 128
`shading interp` 193
Shape 597, 621, 623, 626. *See also* Representation and description
`short` 57
`short e` 57
`short eng` 57
`short g` 57
`showmo` 483
Sifting 112, 255
`sigmamf` 144, 156
`signature` 620
Signatures 619
`single` 26
Singleton dimension 17
`size` 16
Skeleton 623
medial axis transformation 623
morphological 623
`smf` 144
Soft proofing 347
`sort` 431
`sortrows` 604
`sparse` 42
Sparse matrix 42
Spatial
convolution. *See* Convolution
coordinates 13
correlation. *See* Correlation
domain 80, 165
filter. *See* Spatial filters
kernel 110
mask 110, 681
neighborhood 81
template 110, 311, 681
Spatial filtering 109
fuzzy 158
linear 109, 114
mechanics 110
morphological. *See* Morphology
nonlinear 117, 124
of color images 360
Spatial filters. *See also* Spatial filtering
adaptive 233
adaptive median 233
alpha-trimmed mean 230
arithmetic mean 230
average 121
contraharmonic mean 230
converting to frequency domain filters 181
disk 121
gaussian 121
geometric mean 230
harmonic mean 230
iterative nonlinear 246
laplacian 121, 122. *See also* Laplacian
linear 120
log 121
max 126, 230
median 126, 230
midpoint 230
min 126, 230
motion 121
morphological. See Morphology
noise 229
order statistic 124. See also `ordfilt2`
prewitt 121
rank 124. See also `ordfilt2`
sobel 121
unsharp 121
Spectrum. *See* Fourier spectrum
`specxture` 655
`spfilt` 229
`spline` 352
`splitmerge` 585
`sprintf` 60
`sqrt` 64
Square brackets 30, 33, 35, 45
`statmoments` 225
`statxture` 645
`stdfilt` 572
`stem` 96
`strcat` 696

strcmp 73, 697
strcmpi 74, 400, 454, 697
strel 494
Strings. *See* Recognition
strel object 496
stretchlim 84
strfind 698
strjust 698
strncmp 697
strncmpi 698
strread 73
strrep 698
strsimilarity 701
strtok 699
Structure 74
  example 77
  fields 77
  variable 23
Structuring element. *See* Morphology
strvcat 697
sub2ind 40
subplot 384
Subscript 33
sum 37
surf 193
switch 58, 62

T

Template matching. *See* Recognition
text 96
Texture. *See also* Regional descriptors
  spectral measures of 654
  statistical approaches 644
tform structure 279, 345
tofloat 32
tformfwd 281
tforminv 281
tform structure 279
THEN 156
Thresholding. *See* Image segmentation
tic 65
tifs2cv 480
tifs2movie 475
tifs2seq 475
timeit 66
title 96
toc 65
Transfer function. *See* Frequency domain filters
Transformation function. *See* Intensity
transpose 33
trapezmf 143
triangmf 143, 156
true 44, 587
truncgaussmf 145
try...catch 58
twomodegauss 104
Types. *See* Image types

U

uint8 26
uint16 26
uint32 26
unique 604
unravel.c 443
unravel.m 444
Until stability 511
upper 201

V

varargin 88
varargout 88
Vector
  column 13, 15
  norm 245, 675
  row 13, 15
ver 55
version 55
Vertex
  adding noise to 704
  concave 612
  convex 612
  of minimum-perimeter polygon 612
view 191
Vision 2
  computer 3
  high-level 3
  human 3
  low-level 3
  mid-level 3
visreg 309
vistform 283
visualizing aligned images 308

W

waitbar 151
watershed 590
Watersheds. *See* Image segmentation
waveback 409
wavecopy 402
wavecut 401
wavedec2 385
wavedisplay 404
wavefast 391
wavefilter 388
wavefun 382
waveinfo 382
Wavelets
  approximation coefficients 381
  background 377
  custom function 394
  decomposition coefficients 404
    displaying 404
    editing 399
  decomposition structures 396
  downsampling 380
  expansion coefficients 378
  FWTs using MATLAB's Wavelet Toolbox 381
  FWTs without the Wavelet Toolbox 387
  Haar 383
    scaling function 383
    wavelet function 385
    wavelet functions 383
  highpass decomposition filter 380
  image processing 414
    edge detection 414
    progressive reconstruction 417
    smoothing 415
  inverse fast wavelet transform 408
  kernel 378
  lowpass decomposition filter 380
  mother wavelet 379
  properties 379
  scaling 380
  scaling function 379
  support 384
  transform domain variables 377
wavepaste 403
waverec2 409
wavework 399
wavezero 415
wfilters 381
while 58, 60
whitebg 322
whos 17
Windowing functions
  cosine 259
  Hamming 259
  Hann 259
  Ram-Lak 259
  Shep-Logan 259
  sinc 259
Wraparound error. *See* Discrete Fourier transform
wthcoef2 398

X

x2majoraxis 627
xlabel 96
xlim 98
xtick 96

Y

ycbr2rgb 329
ylabel 96
ylim 98
ytick 96
Z
zeromf 145
zeros 44

Z

Zero-phase-shift filters. *See* Frequency domain filters